Linear and nonlinear models for forecasting the realized volatility of cryptocurrencies

Master's Thesis submitted

to

Prof. Dr. Wolfgang Härdle

Prof. Dr. Stefan Lessmann

Humboldt-Universität zu Berlin

School of Business and Economics

Ladislaus von Bortkiewicz Chair of Statistics

by

Ivan Mitkov

(54 2007)

in partial fulfillment of the requirements

for the degree of

Master of Science

in Economics

Berlin, January 20, 2018

Acknowledgement

I would like to thank Junjie Hu for his help during the three months of my research and Bruno Spilak for sharing the high frequency data with me. I am also grateful to all people from Privatissimum Seminar for their ideas and constructive criticism.

Last but not least, an important gratitude to my family and girlfriend for their constant support during my study in Germany.

Abstract

Cryptocurrencies are known for their high fluctuating prices. In order to minimize the risk for investors, many empirical researchers proposed the realized volatility, calculated with intra-daily log returns, as an approximation for the true volatility on the market. In this empirical paper we aim to make possibly the best forecast for the daily realized volatility in the next period and to find a good financial application for our predicted values. For achieving of these goals, some linear and nonlinear models are considered. We make use of the popular linear heterogenous autoregressive model (HAR-RV) and various recurrent neural networks (SRN, LSTM and GRU). Additionally, a hybrid model between the simplest feedforward neural network and the heterogenous autoregressive model is proposed. The results demonstrate superiority of the nonlinear models, whereas this is more obvious for the recurrent neural networks in low volatility times with reduced training data set. The forecasted values are used for calculating the Value at Risk for a portfolio of cryptocurrencies. Unfortunately, the Unconditional coverage test and the Conditional coverage test reject the null hypotheses that the forecasts for future daily realized volatility are a suitable approximation for calculating Value at Risk. The noise and the jumps in the data are pointed as possible reasons for these results, therefore, as a starting point for further research, application of more complex models like HAR-RV-J, HAR-RV-CJ and deep recurrent neural networks is suggested.

Contents

List of Abbreviations

AI	Artificial intelligence	DL	Deep learning
FNN	Feedforward neural networks	GRU	Gated recurrent unit
HAR	Heterogeneous autoregressive model	LSTM	Long short-term memory
ML	Machine learning	NN	Neural networks
RNN	Recurrent neural networks	RV	Realized volatility
SRN	Simple recurrent neural network		

List of Figures

List of Tables

1 Introduction

Digital assets like the cryptocurrencies are known for their high fluctuating prices. However, exactly this fact provoked many people to invest in these digital currencies. The huge amount of money invested and the volume of traded cryptocurrencies challenged also a lot of researchers to focus their empirical works on estimating and predicting the true volatility of such type of assets. Unfortunately, finding a good approximation for measuring the volatility on this certain market turned out to be a difficult task. This step is important, since volatility modeling is mostly very closely linked with risk and uncertainty, thus it is a milestone in asset pricing, portfolio management, option pricing and risk management.

There are various approaches for estimating the true volatility, however, traditionally the most popular way was to calculate the variance of log returns. Later researchers made use of some dynamic volatility estimation models such as the GARCH model or stochastic volatility models. The increasing role of data and the availability of data with intraday returns led to the development of new kind of models based on high frequency data. The first proposal in this direction was made by Merton (1980), who proved that the true variance can be calculated when summing all squared intra-daily returns if the data has sufficiently high frequency. The key concept for this work, namely the realized volatility, was proposed for the very first time by Andersen and Bollerslev (1998), who found that the predictions from traditional models like GARCH are more reasonable when compared to the sum of the squared daily log returns, which corresponds to the realized variance. According to Barndorff-Nielsen and Shephard (2002) the integrated volatility (the true but unknown parameter) could be consistently estimated by the realized variance. However, this is very unrealistic, since prices cannot be continuously observed in practice, thus we end up with biased estimators and noise in the data, which shows some imperfection of the realized volatility as an approximation for the true volatility.

Despite these critiques, there are some stylized facts, that make the realized volatility very attractive for empirical researchers. It is demonstrated that the daily, weekly and monthly measures are characterized with long-range dependencies, i.e. significant autocorrelation in very high lags. Moreover, first, researchers found a negative correlation between the volatility and the asset returns (leverage effects) and, second, it was also shown that the measure includes successfully the jumps from previous period when forecasting future volatility.

Because of these facts many researchers concentrate on predicting correctly the realized volatility and discussing if these forecasts have a good financial application in real practice.

Usually, realized volatility is predicted with the autoregressive fractionally integrated moving average (ARFIMA) models suggested in Andersen et al. (2003) and Thomakos and Wang (2003) as well as with the heterogenous autoregressive model for realized volatility (HAR-RV) proposed by Corsi (2009). The latter became broadly used due to its very simple form of a linear regression, which, however, is still able to reproduce all of the stylized facts cited above. Recent empirical works, e.g Andersen et al. (2007) and Corsi et al. (2012), also suggested the incorporation in the model of jumps from the time series, which resulted it the following extensions - HAR-RV-J and HAR-RV-CJ.

However, financial time series are usually associated with nonlinearity. Usually, it is expected from logarithms or square root of realized volatility to capture the nonlinear element of the data, however, in recent times some nonlinear techniques from machine learning like the neural networks are mostly suggested when dealing with nonlinearity. Baruník and Křehlík (2016) proposed a hybrid model between a simple feedforward neural network (FNN) and HAR-RV called FNN-HAR, whereby the authors used the explanatory variables from the HAR-RV as input variables for their artificial neural network (ANN) and demonstrated an improvement of the forecasts for the next day realized volatility. However, Baruník and Křehlík (2016) are criticized due to the high number of parameters included, which suggests overfitting, therefore Arnerić et al. (2018) reduced the amount of hyperparameters in FNN-HAR and FNN-HAR-J and demonstrated that the competing hybrid models and the classic HAR-RV and HAR-RV-J actually have pretty similar accuracy of the predictions.

The main objective of this paper is to provide possibly best forecasts for the realized volatility of a portfolio from cryptocurrencies and to find a good financial application for the forecasts. For achieving this goal, we make use of the classic linear heterogeneous autoregressive model for realized volatility (HAR-RV), which is known for its memory feature. Moreover, we also propose some nonlinear machine learning techniques for improving the accuracy. To our best known, this is the only paper which tries to involve recurrent neural networks for predicting realized volatility. Therefore, we claim that our empirical contribution completes this specific area of research with something innovative. The simple recurrent neural network (SRN) and its modified architectures - long short-term memory (LSTM) and gated recurrent unit (GRU) are applied for solving the so called for such occasions "regression problem". The selection of these algorithms relies on the fact that they capture the nonlinear element of the data but at the same time they also "memorize" significant events back in the past and use this information while predicting the next outcomes. Additionally, we also reproduce the

proposed by Baruník and Křehlík (2016) and Arnerić et al. (2018) hybrid FNN-HAR model, which should be able to "memorize" information thanks to its input variables equal to those of HAR-RV, but also the model is nonlinear because of the activation function from the feed-forward neural network. When checking the ability of our models to forecast correctly, we differentiate between the scenarios of high volatility and low volatility times. In other words, we compare how good our models "learn" during different volatility times. Furthermore, the ability of the models to "learn" from reduced amount of training data is considered as well. All of the forecasting models are compared through Diebold Mariano test. In the second part of our empirical work we find a financial application for our forecasts and calculate the Value at risk for multiple period ahead. The predicted values are validated through some backtesting techniques.

The empirical results from our work prove the superiority of nonlinear models in terms of forecasting accuracy. This becomes even more obvious if we reduce the training data, when the recurrent neural networks outperform all other models. However, judging by the *p values* from the Diebold Mariano test, these results do not seem to be significant for high volatility times. Considering the financial applications of our forecasts, the null hypotheses for valid Value at Risk forecasts for multiple periods ahead is rejected by both - the Unconditional and the Conditional coverage tests. However, the tendency for better performing recurrent neural networks is confirmed for both - high volatility and low volatility times.

The remainder of the paper is organized as follows. In Section 2 we discuss in detail the concept of realized volatility, Section 3 introduces the linear HAR-RV model and Section 4 gives more information about the theory behind the nonlinear algorithms. The exact training of the models is introduced in Section 5. In Section 6 are provided some descriptive statistics concerning our data and Section 7 covers the actual empirical results from our work. Section 8 concludes.

2 Realized volatility

In recent decades many researchers focused on modeling and forecasting the volatility of financial assets. Reason for this is that volatility is used for many purposes as an approximation for measuring the risk in asset allocation, risk management and option pricing. Since the true volatility is not observable, the struggle to find a good proxy has always been in trend.

Conditional heteroscedasticity and stochastic volatility model cover the most common ways to calculate and forecast the volatility. Unfortunately, these two models heavily depend

on the detection of the corresponding underlying process of volatility, which results in strong restrictions on parameters to be estimated.

Given the growing availability of high frequency data, researchers try constantly to enhance the measures for volatility based on such type of data. The first proposal in this direction was made by Merton (1980), who noticed that the conditional variance can be obtained when summing all squared intra-daily returns if the data has sufficiently high frequency. On the one hand, under some specific assumptions Andersen et al. (2000) and Andersen et al. (2005) demonstrated in their works that using higher frequency data successfully approximate the conditional variance. On the other hand, Barndorff-Nielsen and Shephard (2002) proved that the realized volatility (RV) could serve for estimating the latent variability of the log returns from financial data. So ex post volatility, if the estimations errors and jumps are excluded, becomes observable and could be forecasted by the means of some predicting algorithms.

Let we briefly review the realized volatility in order to prove why it is a good approach for predicting the conditional volatility. Before starting, some notation should be introduced, so, for example, $p(t)$ represents the univariate process of the logarithmic price, r_t is the daily return, and I_t is the entire available information until the time period t.

Suppose,

$$dp\left(t\right) = \mu\left(d\right)d\left(t\right) + \sigma\left(t\right)dW\left(t\right) \tag{1}$$

where μ_t is the drift element and σ is the instantaneous volatility of the process or standard deviation, strictly positive and square integrable, and W_t is the standard Brownian motion (Bucci (2017)).

Let the return between period t and $t-h$ be:

$$r_t = p\left(t\right) - p\left(t-h\right) = \int_{t-h}^{t} \mu_s ds + \int_{t-h}^{t} \sigma_s dW_s \tag{2}$$

with continuous time interval over $[0, T]$ with $0 \leq s \leq t \leq T$. The quadratic variation is:

$$\left[p\right]_t = QV_t = \int_{t-h}^{t} \sigma^2\left(s\right) ds \tag{3}$$

which is the way how the variability to stochastic integration theory is measured (Bucci (2017)).

Equation (3) proves that the drift innovations do not contribute for the volatility of the returns. In such a case the quadratic variation (QV) coincides with the integrated variance

(IV). This is, however, not true in data with jumps and noise. Assuming they are not available in our data, we obtain the following equation:

$$IV_t = QV_t = \int_{t-h}^{t} \sigma^2(s)\, ds \tag{4}$$

According to Andersen et al. (2000) the quadratic variation could be also approximated as:

$$[p(t)] = \operatorname*{plim}_{n\to\infty} \sum_{j=1}^{n} [p(s_j) - p(s_{j-1})]^2 \tag{5}$$

Considering the fact that the daily return is defined as:

$$r_t = \sum_{j=1}^{n} r_{t,i} \tag{6}$$

and that the intraday return looks like

$$r_{t,i} = p_{t,i} - p_{t,i-1} \tag{7}$$

it could be derived that the realized variance (the sum of the intra-daily squared returns), converges in probability to the QV and correspondingly to the IV as long as $n \to \infty$. In other worlds, the realized variance:

$$RV_t = \sum_{j=1}^{n} r_{t,i}^2 \tag{8}$$

leads to

$$RV_t \xrightarrow{p} IV_t \tag{9}$$

Taking the notional volatility into account, which equates the quadratic variation for return series in time interval $[t-h, t]$:

$$\vartheta^2(t,h) = [r,r]_t - [r,r]_{t-h} = \int_{t-h}^{t} = \int_{t-h}^{t} \sigma^2(s)\, ds. \tag{10}$$

one could easily prove that the realized volatility is a consistent estimator of notional volatility, i.e. $RV_t \xrightarrow{p} \vartheta^2(t,h)$. After some recalculations we obtain:

$$E[RV_t | I_{t-h}] \xrightarrow{p} E\left[\vartheta^2(t,h) | I_{t-h}\right] \tag{11}$$

it follows that under specific assumptions (process is square integrable and $\mu(t) = 0$) the realized volatility is an unbiased estimator for the conditional volatility, i.e.

$$E\left[RV_t|I_{t-h}\right] = E\left[QV_t|I_{t-h}\right] = Var\left[r\left(t,h\right)|I_{t-h}\right] \tag{12}$$

where Equation (12) couples the conditional variance from the autoregressive conditional heteroskedasticity - ARCH models with the realized volatility. In other words, one could now develop a model for predicting the conditional volatility using the realized volatility.

Empirical evidence has established the following stylized facts about realized volatility. Firstly, the autocorrelation function is dying out at a hyperbolic rate rather than exponentially which suggests the existence of long memory in the data (Kruse (2006)). Exactly because of this feature we assume that the measure would be an efficient proxy for predicting the volatility of a portfolio from cryptocurrencies and also that it would result in better performance when calculating the portfolio's Value at Risk. Second stylized fact concerning the realized volatility is that its distribution is nearly normal. Third, the distribution is leptokurtic and right skewed - these are another promising facts that a good predicting model for realized volatility would find its financial application in risk management.

Unfortunately, using of the realized volatility as a proxy has also its disadvantages. For example, it is not clear which data frequency is the most suitable, if one wants to make the most accurate prediction, this issue should be considered. However, since in our case only 5 minutes interval data is available, we would concentrate only on this case.

Additionally, the use of realized volatility as a measure for the unobservable fluctuations became especially popular after publishing the work of Andersen et al. (2003), who showed that only in case of daily frequency approaching infinity, respectively the distance between the intra-daily observations going to zero, the realized volatility is a consistent estimator. Since checking if all required criterions for applying RV as an estimator for the conditional volatility goes beyond the scope of this particular work, for the remainder we would assume that these criterions are met and would move to the next section, where we reveal the techniques used for predicting the realized volatility.

3 Linear models

In this section we discuss the most popular linear model for forecasting the realized volatility, namely the heterogenous autoregressive model. Doing this, we aim to give a a good argumentation why the model is included as a benchmark in our work. In the second half of the

section some extensions of HAR-RV are also introduced.

3.1 Heterogeneous autoregressive model

The additive cascade model, going from low to high frequency components, was proposed by Corsi (2009). The main idea of the algorithm is to cover typical features like long memory, fair tails and self-similarity, that are observed in financial time series. At the same time the authors aim to maintain simplicity in the model, which makes its estimations and interpretations much easier.

The economic intuition of the heterogeneous autoregressive model is that different market players take actions based on different temporal horizons of volatility, which refers to the heterogeneous market hypothesis introduced by Müller et al. (1993). Market participants plan their actions based on the trading frequency, which lead to the suggestions that players on the market react differently to diffident temporal components of the volatility. Another finding of Corsi (2009) is that including of further participants actually increase the volatility on the market while, on the other hand, doing this on the heterogenous market causes improvement of convergence. The third fact founded in the paper is that different geographical locations of the market actors contribute also to heterogeneity in the market. Considering these three findings one could conclude that taking the heterogeneous market hypothesis into account, especially in the context of cryptocurrency trading, would be a step in the right direction.

Since the daily realized volatility determines the high frequency return process and the integrated volatility is $\tilde{\sigma}_t^{(d)} = \sigma_t^{(d)}$ wtih $r_t = \sigma_t^{(d)} \epsilon_t$, the model proposed by Corsi (2009) should be able to make predictions for the daily realized volatility in the next period.

The suggested model is a linear function, which considers the partial daily, weekly and monthly volatility. Since the authors concentrate on a typical financial market with 5 trading days for a week and 22 days for a month, the correspondent formulas look like:

$$\tilde{\sigma}_{t+1m}^{(m)} = c^{(m)} + \phi^{(m)} RV_t^{(m)} + \tilde{\omega}_{t+1m}^{(m)} \tag{13}$$

$$\tilde{\sigma}_{t+1w}^{(w)} = c^{(w)} + \phi^{(w)} RV_t^{(w)} + \gamma^{(w)} \mathbb{E}[\tilde{\sigma}_{t+1m}^{(m)}] + \tilde{\omega}_{t+1w}^{(w)} \tag{14}$$

$$\tilde{\sigma}_{t+1d}^{(d)} = c^{(d)} + \phi^{(d)} RV_t^{(d)} + \gamma^{(d)} \mathbb{E}[\tilde{\sigma}_{t+1w}^{(w)}] + \tilde{\omega}_{t+1d}^{(d)} \tag{15}$$

In the equations above $RV_t^{(m)}$, $RV_t^{(w)}$, $RV_t^{(d)}$ represent the monthly, weekly and daily realized volatilities and $\tilde{\omega}_{t+1m}^{(m)}$, $\tilde{\omega}_{t+1w}^{(w)}$, $\tilde{\omega}_{t+1d}^{(d)}$ are the corresponding iid errors terms, $c^{(m)}$, $c^{(w)}$,

$c^{(d)}$ are the intercepts and, finally, ϕ and γ are the coefficients of the explanatory variables. Looking closer, one could notice that all equations consist of an $AR(1)$ part and a second part, which is the expectation for the next larger level from the cascade.

By straightforward recursive substitutions of the partial volatilities and considering the fact $\tilde{\sigma}_t^{(d)} = \sigma_t^{(d)}$, the cascade of equations could be rewritten as:

$$\tilde{\sigma}_{t+1d}^{(d)} = c + \beta^{(d)} RV_t^{(d)} + \beta^{(w)} RV_t^{(w)} + \beta^{(m)} RV_t^{(m)} + \tilde{\omega}_{t+1d}^{(d)} \tag{16}$$

This equation is a three factor model based on the realized volatilities for different frequencies and represents the exact model proposed by Corsi (2009). It can be easily noticed that the left hand side is actually:

$$\tilde{\sigma}_{t+1d}^{(d)} = RV_{t+1d}^{(d)} + \omega_{t+1d}^{(d)} \tag{17}$$

with $\omega_{t+1d}^{(d)}$ being the latent realized volatility and the estimation error. This means that Equation (16) could be reformulated to:

$$RV_{t+1d}^{(d)} = c + \beta^{(d)} RV_t^{(d)} + \beta^{(w)} RV_t^{(w)} + \beta^{(m)} RV_t^{(m)} + \omega_{t+1d} \tag{18}$$

where $\omega_{t+1d} = \tilde{\omega}_{t+1d}^{(d)} - \omega_{t+1d}^{(d)}$.

Equation (18) has a very simple autoregressive form including factors for realized volatilities from different time, therefore it can be labeled as HAR(3)-RV.

Bunch of empirical and simulation studies has shown that the proposed HAR-RV model successfully covers many of the aspects of financial time series. It has been demonstrated that HAR-RV has fat tails, self-similarity and, most importantly for our study, it captures stylized facts as the long memory dependency typical for such kind of data. Additionally, the model proposed by Corsi (2009) shows positive results in covering multifractality, proven in Ma et al. (2014). Despite all these features, the model remains very simple for economic interpretation and calculation because behind hides nothing more than the OLS estimator. Due to these facts we assume that the heterogeneous autoregressive model will be a good model for predicting the $t + 1$ realized volatility.

For realizing the purposes of this particular work we make small modifications of the model, stated in Equation (18). Since the cryptocurrencies have been traded 24 hours a day and 365 days a year, we need to rewrite the regression equation and to assume that a trading day consists of 24 hours, a week of 7 days and a month of 30 days.

3.2 Extensions to the heterogeneous autoregressive model

Various empirical works propose extensions to the heterogeneous autoregressive model. In this subsection some of the most popular models will be briefly presented.

The classic HAR-RV model assumes that the price process is constant. However, in reality it turns out that this process is rather a mixture of a continuous and discontinuous part, therefore the price volatility also consists of two elements - volatility from the continuous part and volatility caused by the jumps, i.e.:

$$dp_t = \mu_t dt + \sigma_t dW_t + \xi_t dq_t \tag{19}$$

with $\xi_t dq_t$ being responsible for the jumps. The quadratic variation of this process takes the form of:

$$QV_t = \int_{t-h}^{t} \sigma^2(s)\, ds + \sum_{t-h<s\leq t} J_s^2$$

So the quadratic variation consists of integrated variance part $IV_{t,h} = \int_{t-h}^{t} \sigma^2(s)ds$ and $\sum_{t-h<s\leq t} J_s^2$ known as the jump variation.

Taking this fact into account, we simply add the jumps J_t to the standard HAR-RV model and obtain its extension called *HAR-RV-J*, which was proposed by Andersen et al. (2007).

$$RV_{t+1d}^{(d)} = c + \beta^{(d)} RV_t^{(d)} + \beta^{(w)} RV_t^{(w)} + \beta^{(m)} RV_t^{(m)} + \beta_J^{(d)} J_t^{(d)} + \omega_{t+1d} \tag{21}$$

Andersen et al. (2007) introduced also another model called *HAR-RV-CJ*, which is based on explicit separation of the realized variance into the continuous and jump components. In other words, the classic explanatory variables of HAR-RV are now replaced by daily, weekly and monthly continuous and jump parts, so the final formula takes the form of the equation below:

$$RV_{t+1d}^{(d)} = c + \beta_c^{(d)} C_t^{(d)} + \beta_c^{(w)} C_t^{(w)} + \beta_c^{(m)} C_t^{(m)} + \beta_J^{(d)} J_t^{(d)} + \beta_J^{(w)} J_t^{(w)} + \beta_J^{(m)} J_t^{(m)} + \omega_{t+1d} \tag{22}$$

HAR-RV-CJ has the advantage that it allows the observation of the exact contribution of each single element regardless if continuous or jump.

The *leverage heterogeneous autoregressive model (LHAR)* was introduced by Corsi and Reno (2009). It extends the heterogeneous structure to leverage effect - this is the larger increase of volatility after a negative shock than after a positive one. This dependency is

modeled with an asymmetric responses of the realized volatility. Another extension is the heterogeneous autoregressive with continuous volatility and jumps (LHAR-CJ) model, which was introduced by Corsi and Reno (2009). The authors identify three main components, that cause the dynamics on the financial market, namely heterogeneity, leverage and jumps, whereas not taking into account of any of these three components contributes to lower forecasting accuracy. The prediction power of the negative past returns has been especially pointed out. Considering all of the listed elements the model is estimated by OLS with Newey-West covariance correction for serial correlation.

$$\begin{aligned} log\hat{V}_{t+h}^{(h)} = c + a^{(d)}log(1+J_t) + a^{(w)}log(1+J_t^{(5)}) + a^{(m)}log(1+J_t^{(22)}) + \\ + \beta^{(d)}logC_t + \beta^{(w)}logC_t^{(5)} + \beta^{(m)}logC_t^{(22)} + \\ + \gamma^{(d)-}r_t^- + \gamma^{(w)-}r_t^{(5)-} + \gamma^{(m)-}r_t^{(22)-} + \varepsilon_t^{(h)} \end{aligned} \tag{23}$$

where J represent the jump component, C stays for the continuous element and r^- marks the negative returns.

A *tree-structured heterogeneous autoregressive (tree-HAR)* process is developed by Audrino and Corsi (2010), whereas realized correlation is equal to the quotient between realized covariances and products of realized standard deviations. According to the authors the model is able to cover two stylized facts of realized correlations, namely the strong temporal dependence and the structural breaks. Audrino and Corsi (2010) demonstrated that the three-HAR also outperforms some classical predicting algorithms like AR(1), ARMA(1,1), ARIMA(1,1,1), a tree-AR(1) and the standard HAR-RV model. The daily tick-by-tick volatility of the *tree-HAR* model takes the following form:

$$\widetilde{RC_{t+1}} = \mathbb{E}_t[\widetilde{RC_{t+1}}] + \sigma_{t+1}U_{t+1} \tag{24}$$

with $\{U_t\}_{t\geq 1}$ being a i.i.d. sequence of the innovations, p_U with expected value zero and variance 1, $\mathbb{E}_t[\cdot]$ is the conditional expectation up to the time t. The conditional dynamics is given by:

$$\mathbb{E}_t[\widetilde{RC_{t+1}}] = \sum_{j=1}^{k}(a_j + b_j^{(d)}\widetilde{RC_t} + b_j^{(w)}\widetilde{RC_t}^{(w)} + b_j^{(m)}\widetilde{RC_t}^{(m)})I_{[X_t^{pred}\in R_j]} \tag{25}$$

$$\sigma_{t+1}^2 = \sum_{j=1}^{k}\sigma_j^2 I_{[X_t^{pred}\in R_j]}, \sigma_j^2 > 0, j = 1, ..., k \tag{26}$$

and $\theta = (a_j, b_j^{(d)}, b_j^{(w)}, b_j^{(m)}, \sigma_t^2 : j = 1, ..., k)$ parameterizes the local HAR dynamics.

The last extension is the *heterogenous autoregressive with gamma and leverage (HARGL)* model is shown by Corsi et al. (2013). The authors suggest a discrete-time stochastic volatility option pricing model, which uses the gained information from the realized volatility. The estimated RV serves as a proxy for the unobservable volatility of the log returns. The empirical work proves that two factors are especially important for the performance of the model, namely the use of realized volatility, which provides a good and fast-adapting proxy for the unobserved volatility, and, on the other hand, the high persistence and smoothing caused by the HARGL model. Due to these features of the algorithm, HARGL is better in reproducing the $\mathbb{Q}$-dynamics, hence in achieving better results than the typical GARCH option pricing models. One should take into consideration that the HARGL model differs from the classic HAR-RV in the way the explanatory variables have been calculated, namely:

$$RV_t^{(t)} = \frac{1}{4}\sum_{i=1}^{4} RV_{t-i}^{(d)} \tag{27}$$

$$RV_t^{(t)} = \frac{1}{17}\sum_{i=21}^{5} RV_{t-i}^{(d)} \tag{28}$$

Doing this, prevents overlapping of volatilities for the various periods.

Finally, one could conclude that the various extensions of the HAR-RV model and their successful application with real world data prove the predicting power of the algorithm.

4 Nonlinear models

When explaining the methodology hiding behind this hybrid model, we firstly need to introduce a popular algorithm from machine learning (ML), namely the artificial neural network (ANN) and more concretely the feedforward neural network (FNN). Since FNN and recurrent neural networks (RNNs) share huge amount of similarities, in the next subsection we introduce important definitions and parameters, which will be valid for the nonlinear algorithms in this particular work.

In recent years, in order to capture nonlinearity in the forecasting models, more attention has been given to some techniques from machine learning and more specifically to the neural networks (NNs).

In this section we discuss different architectures of artificial neural networks. We explain how they are constructed and how their application contributes for solving our problem. Since we are looking for models combining both - nonlinearity and the memory feature, a natural

suggestion is the implementation of the recurrent neural networks. Due to the fact that we develop also an hybrid between FNN and the HAR-RV, we have to reveal the theory behind the most simple ANN as well.

The section is divided as follows. We start introducing the classical feedforward neural network and afterwards we discus in detail the recurrent neural networks and more precisely its three most suitable architectures proven to be the most efficient NNs for empirical works with financial time series.

4.1 Deep feedforward neural networks

Deep feedforward neural networks or also called multilayer perceptrons (MLP) gained more attention in recent years. This could be explained because of the fact that the volume of data used is increasing dramatically, the performance of the modern computers improves constantly and most importantly - these nonlinear techniques perform pretty good in context of predictive analytics. The FNN emerged from a very popular machine learning algorithm called Perception, which was developed by Rosenblatt (1958) and inspired by McCulloch and Pitts (1943).

The idea behind the model is the approximation of some function f^*, like, for instance, $y = f^*(x)$, where the function f^* maps the input variable x with the target values y. What exactly the FNN does, is to define the mapping $y = f(x;\theta)$ and to find the value of θ, resulted from the best approximation.

The function is known as feedforward because the information flows only one way - from the function x, though the function f and then it achieves the target values of y. In other words, there are no feedback connections as in the back-propagation algorithm (such type of neural networks will be discussed later).

Neural networks are called like this because they are typically represented by combining many different functions. The visualization of the model with all its inputs, levels, connections and outputs resembles the structure of a biological neural network. For instance, one could have three various function, let we say $f^{(1)}$, $f^{(2)}$, $f^{(3)}$, and these functions are coupled into a chain $f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}\right)\right)$. Translated to the language of ML, we have a neural network with three layers, whereby the first function $f^{(1)}$ represents the input layer, containing all input values, $f^{(2)}$ is the second layer - in our case the hidden layer, and the last one is the output layer. The length of this chain shows the *depth* of our particular NN.

Finally, the dimension of the hidden layer defines the exact *width* of the model. Each unit

in this layer could be associated with the role of a neuron in the neuroscience, therefore the units are also called neurons.

One way to understand feedforward networks is to begin with linear models and then to consider how the problem of nonlinearity could be solved. This is where NNs could be efficiently used. To transform these linear models in nonlinear functions of x, one could make use of the nonlinear model not directly to x but to $\phi(x)$, where ϕ stays for a nonlinear conversion.

However, in this step probably one would ask how exactly should the ϕ be formulated. The idea of deep learning is to learn ϕ. In the neural networks' approach we obtain the model:

$$y = f\left(x; \theta, \omega\right) = \phi\left(x; \theta\right)^{\top} \omega \tag{29}$$

where θ is used to learn ϕ, ω maps the $\phi\left(x\right)$ to the target value and ϕ could be interpreted as the hidden layer. Essentially, we parametrize the transformed linear models as $\phi(x; \theta)$ and then apply an optimization algorithm (discussed in detail in Section 5.2) to estimate a θ, which is associated with a good representation.

After introducing the main idea behind deep feedforward neural networks, we discuss some of the main concepts in detail below.

Single layer perceptron vs. multilayer perceptron

A *single layer perceptron (SLP)*, shown in Figure (1), is a feedforward network based on a threshold transfer function. SLP is the simplest ANN, which could classify only linearly separable variables with a binary target $(1, 0)$. Output is then obtained thanks to the following formula:

$$output = \begin{cases} 1 & \sum w_i x_i > \theta \\ 0 & else \end{cases} \tag{30}$$

where w_i stays for weight of the *i-th* input x.

The SLP, however, does not know the initial weight w_i, therefore it is randomly assigned at the beginning. The algorithm sums then all the weighted inputs and if the result is larger than a certain threshold θ, then the SLP neural networks is activated, i.e. output is equal to 1, otherwise it is 0 (see Equiation (31)). As introduced above, the SLP works properly only if the cases are linearly separable.

$$\begin{aligned} w_1 x_1 + w_2 x_2 + ... + > \theta \rightarrow Output = 1 \\ w_1 x_1 + w_2 x_2 + ... + < \theta \rightarrow Output = 0 \end{aligned} \tag{31}$$

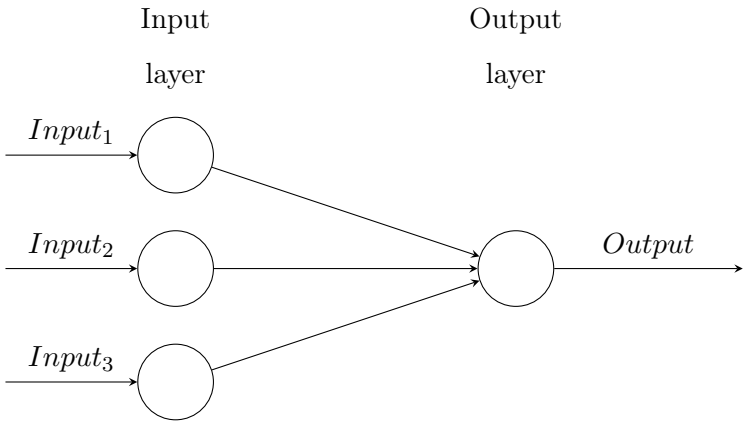

Figure 1: Single layer perception.

A very common example for the inability of SLP to solve problems with linearly non-separable data is the *XOR function* or the "exclusive or" problem, which could be solved by the means of *multilayer perception (MLP)* using the *back-propagation* algorithm. However, let we firstly introduce this traditional example.

The XOR function is the problem of using neural networks, in which we have exactly two dummy variables as inputs. The XOR function should return exactly 1 if at least one of the input variables is equal to 1 and 0 otherwise. Expressed differently, we want to learn our target function $y = f^*(x)$ and our model proposes $y = f(x;\theta)$ with θ making f as close to f^* as possible.

In the next step we have to minimize our loss function $Q(\theta)$. For simplicity Goodfellow et al. (2016) suggested the MSE, although it is not the best loss function, when it comes to dummy variables.

$$Q\left(\theta\right) = \frac{1}{4}\sum_{x\in\mathbb{X}}\left(f^*\left(x\right) - f\left(x;\theta\right)\right)^2 \tag{32}$$

Supposing we choose a linear model for $f(x;\theta)$, we obtain the following function:

$$f\left(x;\omega,b\right) = x^\top\omega + b \tag{33}$$

The linear model gives 0.5 as a result everywhere. To solve this problem, we apply the simple feedforward network with one hidden layer with hidden units h, computed by $f^{(1)}(x,W,c)$. These hidden units are then used as inputs to the output layer. So, at the end we still do have the linear model but now the output is produced by the hidden units h and not by the raw input values x. The whole model obtains the following form $f(x;W,c,\omega,b) = f^{(2)}\left(f^{(1)}\left(x\right)\right)$.

This means that for $f^{(1)}$ being linear the whole model remains linear. What we need is the activation function g to transform the features. This leads to:

$$h = g\left(W^\top x + c\right) \tag{34}$$

with W being the weights of the linear transformation and c the bias. One should take into consideration that the activation function is normally chosen to be applicable for each element, which yields to:

$$h_i = g\left(x^\top W_{:,i} + c_i\right) \tag{35}$$

Later in Section 5.2 we introduce various activation functions and discuss their advantages and disadvantages in detail. For this particular case if we make use of the popular ReLU activation, the whole neural network will look like:

$$f\left(x; W, c, \omega, b\right) = \omega^\top max\left\{0, W^\top x + c\right\} + b \tag{36}$$

Taking Equation (36) into account, it easily could be estimated that:

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{37}$$

$$c = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{38}$$

$$w = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \tag{39}$$

with $b = 0$.

Considering X as an input matrix, established thanks to both dummy variables:

$$X = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \tag{40}$$

and replacing all parameters from Equation (36) with the values from Equations (37), (38), (39), we obtain exact our target values.

However, this is a very simple example, in practice there might be thousands of parameters, which, of course, leads to larger complexity and deviations between forecast and target values. For example, we could have a neural network with more than one hidden layer. This would change the whole model to:

$$\begin{aligned} h^{(1)} &= g^{(1)}\left(W^{(1)\top}x + b^{(1)}\right) \\ h^{(2)} &= g^{(2)}\left(W^{(2)\top}h^{(1)} + b^{(2)}\right) \\ &... \\ h^{(depth)} &= g^{(depth)}\left(W^{(depth)\top}h^{(depth-1)} + b^{(depth)}\right) \end{aligned} \tag{41}$$

When an ANN has two or more hidden layers, it is called a deep neural network (DNN) (see, for example, Figure (2)). The problem of the neural network's dimensions, i.e. width and depth, was the focus on many empirical works. In practice this question is not that simple and depends on the data and what we aim to do with it. This issue is discussed in detail in Section 5.2. However, Lippmann (1987) showed that a MLP with only two hidden layers is sufficient for creating classification regions of any desired shape. This is instructive, although it should be noted that no indication of how many nodes to use in each layer or how to learn the weights is given. On the other hand, Alvarez and Salzmann (2016) proved that the number of neurons could be reduced by 80% and the accuracy of the prediction is still roughly the same. This corresponds to the aimed for our purposes very simple architecture of neuronal networks. Before moving to the next type of NNs and the actual learning of the networks, we have to illustrate another important concept, namely the back-propagation.

Back-propagation algorithm

The groundbreaking article of Rumelhart et al. (1986) introduced the back-propagation training algorithm for the very first time. For each training instance the algorithms feeds it to the network and then estimates the output of every single neuron in every single layer. After calculating the networks's output, we have to find out the error between the forecasted value and the desired value. This is possible thanks to a specific for the problem loss function (we will introduce various types of loss functions in Section 5.2). In the next step, the back-propagation algorithm finds out how much each neuron in the last layer contributed to each output neuron's error. It then proceeds like this layer by layer until the algorithm reaches the input layer. Namely because of this backward step the algorithm is defined as back-propagation. In this phase we estimate the error gradient across all connection weights

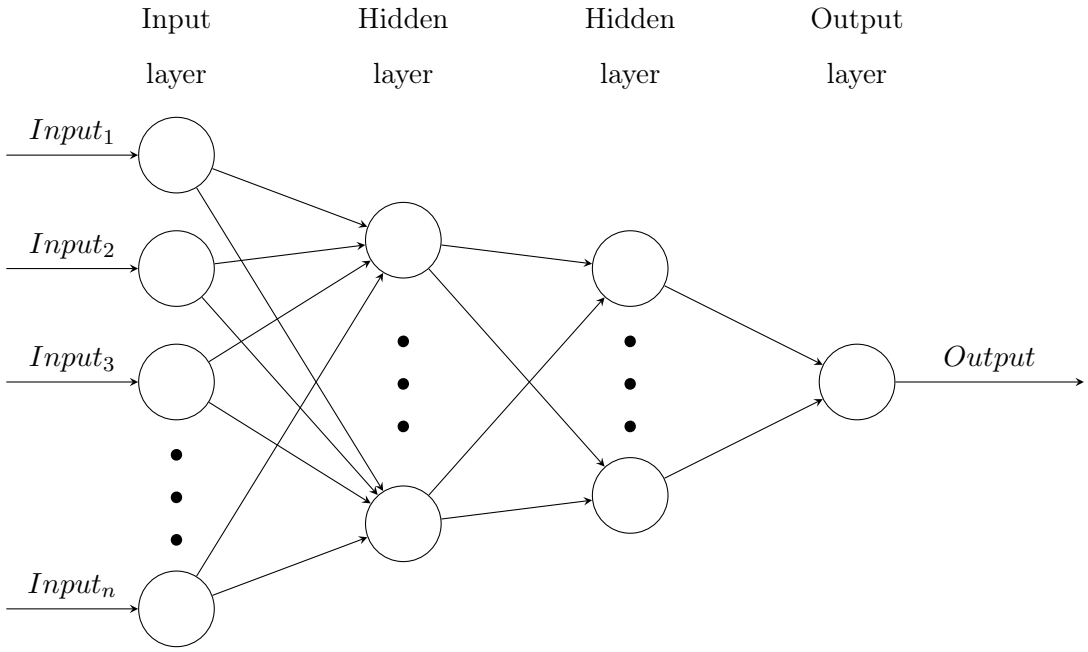

Figure 2: Deep neural network with two hidden layers.

by propagating the error in reverse (backward) direction. At the end of the back-propagation algorithm is the *gradient descent* step. For realizing of its calculations we use the error gradients, that were already measured. *Gradient descent* is, in other words, the engine hiding behind the back-propagation algorithm and making the learning process possible. This term will be explained in detail in Section 5.2.

4.2 Recurrent neural networks

Recurrent neural networks are another family of ANNs. They are formed by taking a feedforward network and adding an additional connection to the previous layers. The already introduced back-propagation algorithm also successfully trains these networks, however, the patterns must always be in a sequential form. Unlike feedforward neural networks, RNNs have an extra neuron that is connected directly with the hidden layer just like the other input neurons. This additional element holds the information of one of the layers as it was there at the training of the NN also in the previous time step. This extra neuron is called the *context unit* and contributes for the long-term memory feature of the recurrent neural network. Due to this property of the model, it is increasingly important for predicting financial time series, on the one hand, and language and pictures recognition, on the other hand.

In order to make the whole picture clear, we consider different architectures of RNNs and discuss why they are suitable for predicting the $t + 1$ values of such a dynamic process like

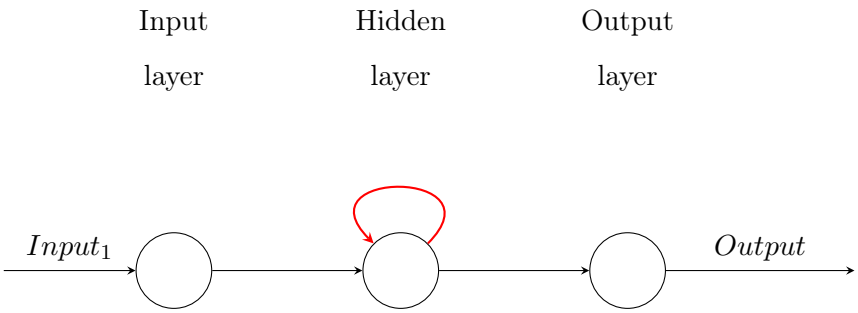

Figure 3: A simple recurrent neural network

the daily realized volatility of cryptocurrencies.

4.2.1 Simple recurrent neural network

The term Simple recurrent network (SRN) often refers to the network architectures proposed by Jordan (1986) and Elman (1990).

An Elman network is illustrated in Figure (3) and has three layers and the new element context units. These units are connected with the hidden layer and have a constant weight of one (the idea of fixed-weight recurrent edge is fundamendtial for the long short-term memory networks, however, this would be introduced in detail in the corresponding section). At each learning step the input values are fed-forward making use of the learning rule proposed in the former section. It should be pointed out that the back-connections firstly make copy of the previously seen values of the hidden units. This information is used in the next step of the learning algorithm. Thus, the network can maintain information, which makes it very useful for problems with sequencial data. The Emlan's network is equivalent to a simple RNN, in which each hidden node has a single self-connected recurrent edge.

$$\begin{aligned} h_t &= \sigma_h \left(W_h x_t + U_h h_{t-1} + b_h \right) \\ y_t &= \sigma_y \left(W_y h_t + b_y \right) \end{aligned} \tag{42}$$

The networks introduced by Jordan (1986) are very similar with the only difference that the hidden layer is extended by so called *state units*. The values from the output nodes are fed to the special units, which afterwards feed these values to the nodes of the hidden layer at the next time step $t + 1$. Additionally, the state units are self-connected and contribute for sending information across multiple time steps without perturbing the output at each intermediate time step (Lipton et al. (2015)).

Recurrent connections from the state units to themselves and from the output to the state units make possible that the output from period t is used as input in period $t + 1$. Figure

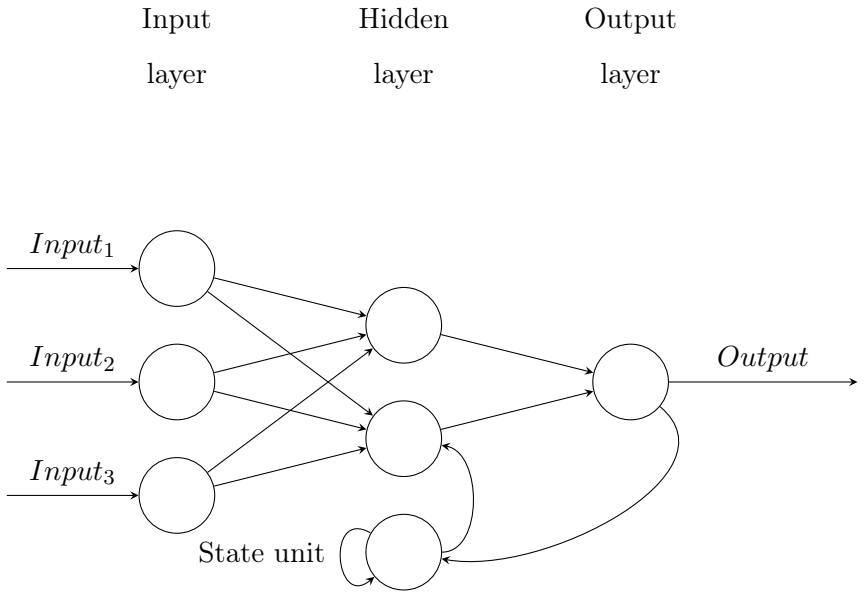

Figure 4: A simple Jordan recurrent neural network.

(4) represents a simple Jordan's network with one hidden layer and out neuron in the output phase.

$$\begin{aligned} h_t &= \sigma_h \left(W_h x_t + U_h y_{t-1} + b_h \right) \\ y_t &= \sigma_y \left(W_y h_t + b_y \right) \end{aligned} \tag{43}$$

In equations (42) and (43) the corresponding parameters are as follows:

$$\begin{aligned} x_t &: Input\ vector \\ h_t &: Hidden\ layer \\ y_t &: Output\ vector \\ W, U, b &: Parameter\ matrices \\ \sigma_h, \sigma_y &: Activation\ functions \end{aligned}$$

As noted above, the recurrent neural networks became especially attractive in the field of finance due to their ability to capture nonlinearity and to solve the long memory problem, which is typical for financial time series. So, for example, Lawrence (1997) compared the predictions from a simple recurrent neural network to some statistical and regression techniques and showed that RNN is able to predict 92% of the price movement correctly and the benchmark models only performed at a 60% rate. Wang and Leu (1996) additionally

researched the prediction power of simple recurrent neural networks. They put the factors from an $ARIMA(1, 2, 1)$ in the proposed NN and demonstrated that the model is able to deliver acceptable accuracy of the predictions for period up to 6 weeks ahead.

RNNs find application in volatility forecasts as well, however, their application for predicting the intraday realized volatility is something, that is not researched that intensively. Liu et al. (2018) demonstrated that the RNN have very similar results compared to HAR-RV-J but the nonlinear model need much shorter input time frame. This signals that if the historical data is scarce, we could rely on the model from the field of the modern AI. Furthermore, the errors from the RNN are uniformly lower, while those from the linear model reduce only in case of larger historical data set. The authors also tried to find some financial application of their predictions and proved that their model contributes for an attractive Sharpe ratio for trading a volatile derivative.

However, for training recurrent neural networks on long sequences like those, associated with the financial time series, the model has to run over many time series, which causes a very deep neural network. This is especially relevant for our case problem.

From the previous section we are familiar that the updated weight in the back-propagation is the product from the multiplication of the learning rate, error term of the previous layer and the input in this particular layer. When applying an activation function (we will regard the most popular functions later) like the sigmoid, the small values of its derivate get constantly smaller, which results in vanishing of our gradient, as we move to the starting layers. Thus, it becomes difficult to train the starting layers and the algorithm never converges to a good solution. The problem is called *vanishing gradients*. There is also another case rather encountered in recurrent neural networks, when the gradients grow bigger and bigger, so many layers get larger weights. This corresponds to a state called *exploiding gradients* problem. There are some tricks to overcome these two problems like good parameter initialization, faster optimizers and non-saturating activation functions (see Section 5.2 for more). However, if the RNN needs to handle even moderately longs sequences (e.g., 100 inputs), then training will still be very slow (Géron (2017)).

For solving these issues various types of architectures of cells with long-term memory were suggested. Most popular among empirical researchers got the long short-term memory cell and the gradient recurrent unit. They demonstrated far better results than the simple recurrent neural networks, so in recent years the basic cells have not found an application anymore. Because of these facts, we aim to make use of the newly proposed cells in order to

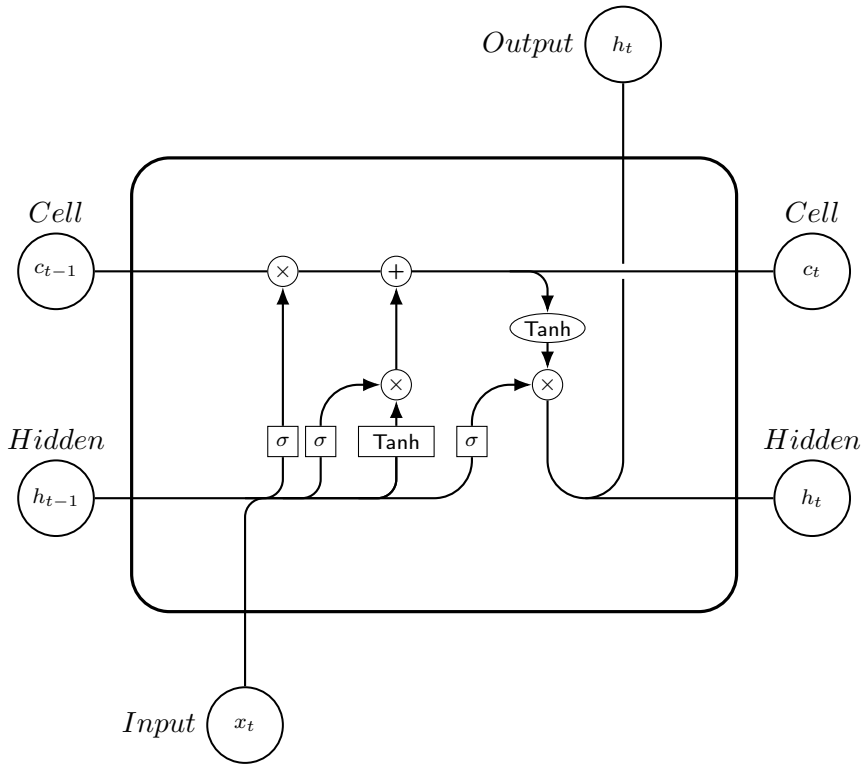

Figure 5: Architecture of LSTM.

predict in the best way the realized volatility, therefore we regard both architectures in detail in the next two sections.

4.2.2 Long short-term memory

As noted above, in order to solve the problem of *vanishing/expoiding gradients* researchers developed the long short-term memory model (LSTM), which has gained in importance in recent years.

The algorithm was proposed by Hochreiter and Schmidhuber (1997) and has been enhanced over the years by several researchers such as Zaremba et al. (2014) and Graves and Schmidhuber (2005). Basically, LSTM could be considered like a normal cell with the difference that its performance will be much better. The time for estimation and convergence will be significantly improved and, most importantly for our work, the model is explicitly designed to avoid the long-term dependency problem.

All RNNs are known with their form of a chain of repeating modules of neural network. This standard chain is represented in Figure (3). LSTM has this structure as well, however, instead the single neural network layer, there are four layers, interacting very specifically. A typical LSTM network (see Figure (5)) is composed of different memory blocks called cells. It

is important that there are two states, which have to be transferred to the next cell (let we say from the cell in $t-1$ to the cell in t). These states are known as the *cell state* and the *hidden state*. The memory blocks build three mechanisms called gates, which allow memorizing things and manipulations with this memory. Each of these important components of LSTM is being discussed below.

Cell state or *memory cell* is the key of LSTM. It runs through the whole neural network chain, whereas only some small linear corrections are undertaken, which contributes for the flow of a constant error through the memory block. It should be pointed out that the information of a given cell state could be characterized with three different dependencies. They could be generalized like this - firstly, we have the information from the previous step or the previous cell state (in our case the memorized realized volatility from the former period), secondly, the output from the previous cell known also at the previous hidden state and, finally, the input from the current time period t.

However, as discussed above, the model is able to remove or add information to the cell state, which is possible thanks to the gate cells. This structure acts to the received information and decides if to pass or to block the information based on its given weight. The gate cells are defined below.

The *forget gate* is responsible for deleting information from the cell state. This information, which does not have much importance or is not required any more, is deleted via a multiplication or a filter. Due to this feature of LSTM the performance of the model is improved. The forget gate requires two inputs - h_{t-1} and x_t, which are respectively the hidden state from the previous period and the input value from the current period t. Both, the h_{t-1} and the x_t, are multiplied by the weight matrices and a bias is added. Following this, a *sigmoid function* with values from 0 to 1 is applied to the product. Basically, this function contributes for deciding which values to keep and which to pass, whereas a 0 means that the forget gate wants to remove this value, and a 1 corresponds to memorizing the given information. This vector is then multiplied to the cell state.

The *input gate* plays the main role for the addition of new information to the cell state. This is essentially a three-step process. At the beginning it should be decided what values have to be entered to the cell state by involving a sigmoid function. Afterwards, a vector containing all the possible information is created. This is done by the means of a *tanh* function, which gives values ranging form -1 to 1. At the last step the values from the sigmoid function are multiplied with the vector (values from tanh function) and then the useful information should

be put in the cell state.

The task of selecting which information from the current cell state should be selected and shown out as an output is undertaken via the *output gate*. This could be again considered as a three-step process. At the first phase with the help of the tanh function all values from the cell state are scaled with values from -1 to 1. Afterwards a filter using the values of h_{t-1} and x_t and the sigmoid function is applied in order to choose what exactly should be given as an output. Finally, these both vectors are multiplied with each other and the information is then transformed to output but also sent to the next cell.

When we go through the whole architecture step-by-step, we deal with the following equations:

$$f_t = \sigma\left(W_f\left[h_{t-1}, x_t\right] + b_f\right) \tag{44}$$

$$i_t = \sigma\left(W_i\left[h_{t-1}, x_t\right] + b_i\right) \tag{45}$$

$$\tilde{C_t} = tanh\left(W_C\left[h_{t-1}, x_t\right] + b_C\right) \tag{46}$$

$$C_t = f_t \times C_{t-1} + i_t \times \tilde{C_t} \tag{47}$$

$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right) \tag{48}$$

$$h_t = o_t \times tanh\left(C_t\right) \tag{49}$$

with σ being the sigmoid function, W - the weighting matrix, h - the hidden state, x - the current input, b - the error term, $tanh$ - tanh activation function, C - the cell state and f, i and o being respectively the forget, the input and the output gates.

To our best knowledge this is the first attempt for using the long short-term memory model for forecasting the realized volatility of the next period. The algorithm, however, finds broad application in predicting stock prices and log returns, where the major of the researchers prove the model's power in forecasting the corresponding values. With increasing the popularity of machine learning techniques increases also the number of attempts for predicting the stock prices and volatilities by the means of these algorithms. So, for example, McNally et al. (2018) showed that LSTM achieves better accuracy in predicting the direction of bitcoin's

price, while competing with an autoregressive integrated moving average model (ARIMA). Additionally, Yu and Li (2018) also demonstrated the superiority of LSTM over other model in predicting the volatility of Shanghai Compos stock price index. By comparing the values of four types of loss functions, we illustrate that LSTM model has a better predicting effect. These results and the fact that, to our knowledge, there are no other attempts in predicting realized volatility with LSTM, convinced us for including the algorithm among our models.

We introduced the most basic LSTM algorithm, in fact, almost every empirical work related to this model suggests slightly different variation defined by the authors as an improvement of LSTM. In our opinion, only the Gated recurrent unit is worth it to be discussed in detail for the purposes of our work, therefore it is regarded separately in Section 4.2.3.

4.2.3 Gated recurrent unit

Another efficient way for solving the *gradient vanishing* problem is the gated recurrent unit (GRU). The algorithm was proposed by Cho et al. (2014) in order to make each recurrent unit able to capture long-term dependencies. Like the LSTM the network consists of gating units, that control the flow of information, however, without having special memory cells in the architecture.

In contrast to the LSTM, that uses forget, input and output gates, the GRU operates using a reset and an update gate. The reset gate is situated between the previous activation and the next candidate activation to remove the previous state, on the other hand, the update gate selects how much of the candidate activation should be used for updating the cell state.

GRU is visualized in Figure (6), where r stays for the reset gate and z is the update gate. Intuitively, the reset gate regulates how to couple the new information with the previous one, and the update gate determines which information from the previous time period should be kept. The GRU could be regarded also as a specific vanilla case of the RNN if the reset is set to all 1's and the update gate to 0's.

Let we go through the whole architecture step-by-step. One starts calculating the update gate z by the means of the following equation:

$$z_t = \sigma\left(W^z x_t + U^z h_{t-1}\right) \tag{50}$$

where x_t is multiplied by its own weight W^z, h_{t-1} contains the information from the former period and is also multiplied by its weights U^z. In the next step both results are added to each other and multiplied by the sigmoid activation function, which leads to a product between 0

and 1.

Afterwards, GRU selects which information should be maintained, this is possible thanks to the reset gate, which makes use of the formula bellow.

$$r_t = \sigma \left(W^r x_t + U^r h_{t-1} \right) \tag{51}$$

It resembles Equation (50) with the difference coming from the weights and the main object of the this gate.

Now, let we see how exactly both gates will reflect on the output. First, the new memory content should be detected and stored next to the important information from the past. This procedure is completed from the reset gate, which uses the following equation:

$$\tilde{h}_t = tanh \left(x_t U^t + \left(r_t * h_{t-1} \right) W^h \right) \tag{52}$$

After multiplying the input with its weight and adding the element-wise product of the reset gate and $h_{t-1} W^h$, which determines what should be removed from the previous period, we sum both results and multiply them with the nonlinear tanh activation function.

Finally, we have to calculate the information vector h_t, that should be passed down to the network. At this point the update gate is included. It decides what should be kept from the current memory content $\tilde{h}_t$ and what from the former period h_{t-1}. This process follows the equation below:

$$h_t = (1 - z_t) * h_{t-1} + t_t * \tilde{h}_t \tag{53}$$

The efficiency of GRU is the focus of Chung et al. (2014), who evaluated the model on the tasks of polyphonic music modeling and speech signal modeling. The authors concluded that both, the LSTM and GRU outperform significantly the classical RNN, however, both modifications - LSTM and GRU do not differ from each other significantly. Di Persio and Honchar (2017) compared the model to the outcomes from RNN and LSTM in order to make forecasts of Google assets. The authors demonstrated that GRU perform slightly better than the other algorithms for smaller training data sets. This property of the model makes us optimistic, when we try to forecast the realized volatility of such a higher asset as the cryptocurrencies.

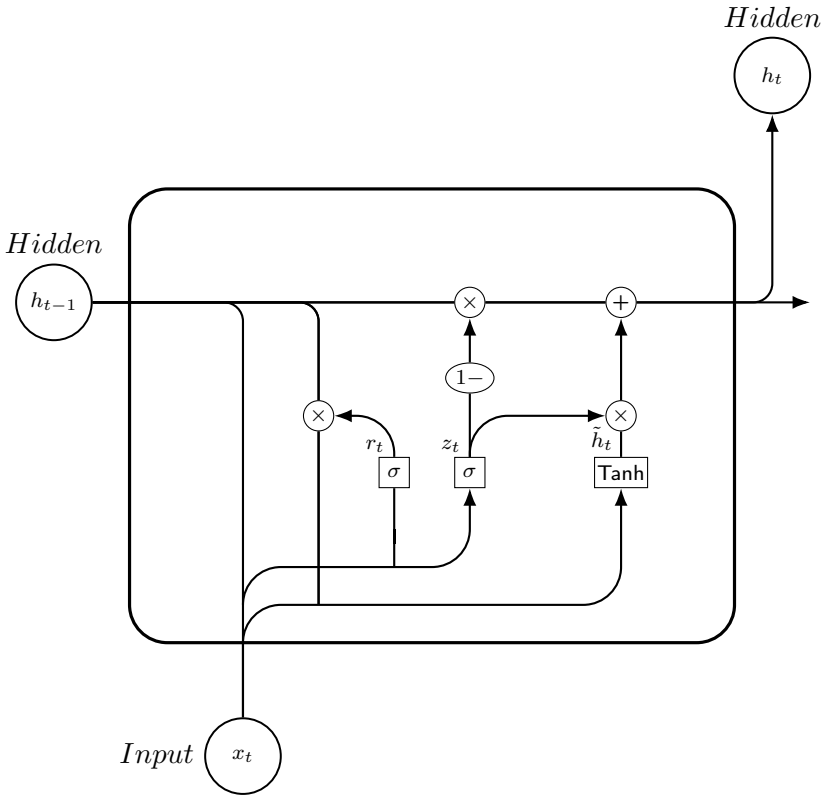

Figure 6: Architecture of GRU.

4.3 A hybrid between neural networks and HAR

Artificial intelligent techniques are especially valuable, when it comes to making predictions with extremely volatile time series data. Unfortunately, their application for forecasting the future realized volatility has been evaluated only in few papers. Therefore, in our current work we aim to contribute with new results to this area of research. For that goal we concentrate mainly on the empirical work of Arnerić et al. (2018), where the authors build a hybrid model between the classical HAR-RV and the feedforward network (FNN). For their estimation Arnerić et al. (2018) took the explanatory variables from HAR-RV and defined them as inputs in a FNN.

The authors of the model proved that, on the one hand, when it comes to in-the-sample prediction accuracy, the FNN-HAR model slightly outperforms the benchmarked HAR-RV and HAR-RV-CJ, although, based on a Diebold Mariano test, the differences are not significant. On the other hand, the classical HAR-RV models behave better in out-of-sample predictions. It should be additionally said, that Arnerić et al. (2018) do not use the full potential of NNs and intentionally do not include large number of neurons and hidden layers, since it will lead to overfitting and distortions of the accuracy measurements.

A very similar model finds usage in Baruník and Křehlík (2016), where different time lags of the oil prices entered the FNN as input variables and contributed for the predictions of the prices in $t+1$. After comparing the predictions from the proposed FNN-HAR with the forecasts from HAR-RV and ARFIMA, it turned out, that the hybrid FNN-HAR model yielded to the lowest error uniformly through all tested periods. It is also shown that the better performance led to economic benefits, especially when calculating the future Value at Risk. According to the authors working with the median realized volatility, proposed by Andersen et al. (2012), is another factor, that contributes for better results. The equations takes the following form:

$$\widehat{IV}_{t,h}^{(MedRV)} = \frac{\pi}{6 - 4\sqrt{3} + \pi} \left(\frac{N}{N-2} \right) \sum_{k=3}^{N} med \left(|\Delta_{k-2,y_t}|, |\Delta_{k-1,y_t}|, |\Delta_{k,y_t}| \right)^2 \tag{54}$$

where Δ_{k,y_t} stays for the price jumps of size Δj_i and N are the number of intraday observations.

Capturing the proposed hybrid models from Arnerić et al. (2018) and Baruník and Křehlík (2016), we end up with a new algorithm being able to reproduce the memory feature of financial series but also having a nonlinear character thanks to the activation function of the NNs. In order to prevent the model from overfitting and to guarantee an equal environment for all competing models, we maintain our neural network with a very simple structure. However, the exact selection of hyperparamers and training of the algorithms is explained in detail in the following section, therefore let we move forward.

5 Estimating and training the predicting models

In this particular section we would regard how all of our models are trained. Additionally, we discuss briefly how the parameters fine tuning happen and also reveal some of the theory accompanying the hyperparameters, especially those of neural networks. The corresponding empirical results are summarized in Section 7, whereby we have to say that all calculations within the work are done in *Python*. We mostly made use of its modules *scikit-learn* and *keras* respectively for HAR-RV and the neural networks.

5.1 Estimating HAR-RV

The heterogeneous autoregressive model of the realized volatility takes the form of simple linear regression. Since in the standard model proposed by Corsi (2009) the authors work with

assets, traded on standard stock markets, they assume that a trading day consists of 8 hours, a week of 5 days and a month of 22 days. However, the exchange market for cryptocurrency does not have such restrictions, since it is traded 24 hours a day and 365 days a year. Due to these facts, we need to modify Equation (18). The new linear regression obtains the following form:

$$RV_{t+24h}^{(24h)} = c + \beta^{(1d)} RV_t^{(1d)} + \beta^{(7d)} RV_t^{(7d)} + \beta^{(30d)} RV_t^{(30d)} + \omega_{t+24h} \tag{55}$$

RVOLharfnn

where $RV_t^{(1d)}$ stays for the realized volatility of a time window from $24h$, $RV_t^{(7d)}$ is the realized volatility for $7 \times 24h = 168h$ and the monthly realized volatility from the original Equation (18) is replaced by $RV_t^{(30d)}$, which corresponds to $30 \times 24h = 720h$. Once again ω_{t+24h} stays for the error term. The exact estimations from the model could be found in the *Quantlet* cited above.

After using the training data set for estimating the coefficients from the linear regression of Equation (55) with the standard ordinary least squares (OLS), we use the information from the test data to make our predictions.

5.2 Training and tuning neural networks

Since training a neural network combines hyperparameters and steps, which are common for the types of NNs, used in our empirical work, we would regard in detail the procedure in this particular section step-by-step and then give reason for choosing specific parameters.

Training a neural network belongs to the algorithms associated with the *supervised learning*. As a supervised learning is described the procedure with an input data set $\mathbf{X}$ and an output vector or matrix $\mathbf{Y}$. The goal of this procedure is to find a mapping function to predict the output thanks to the given information in the input data. This type of learning is called supervised because it resembles the teacher from school, who corrects the students for false answers or, in our case, the algorithm is corrected after its forecasts with large deviations from the target value (the true realized volatility in $t + 24h$).

Supervised learning splits in two main groups:

- *Classification* - this is the case when the output variable should be a category, for example, "blue", "red" and "green" or "stock price raises" and "stock price falls".
- *Regression* - is the situation when the result is a real value, e.g. the weight of an object

or stock price in the next period. Thus, trying to predict the realized volatility in $t+24h$ naturally leads us to this class.

Loss functions

In order to correct the errors in prediction, made by the algorithm, we should specify the *loss function*. In other words, we need a function $L(x, y, \theta)$ (Goodfellow et al. (2016)) that has to measure the quality of our prediction $\hat{y}$. The loss functions could also be divided into two groups for classification and regression problems. However, since we work only on predicting exact values, only the associated with this case loss functions are considered here.

One of the most common for such occasions functions, available also in the extension of *TensorFlow - Keras*, are summarized here:

- *Mean squared error* - is the most commonly used function for regression cases. It is simply the sum of squared distances between the target and the forecasted values. MSE takes values between 0 and ∞.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_t})^2$$

- *Mean absolute error* - equals the sum of absolute differences between target values and predictions. In other words, it calculates the absolute deviation from the target but it does not consider the direction of the deviation. The value also ranges from 0 to ∞.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y_t}|$$

- *Mean absolute percentage error* - usually expresses the accuracy as the percentage error, whereby it uses the following formula:

$$MAPE = \frac{100\%}{n}\sum_{i=1}^{n}\frac{y_i - \hat{y_t}}{y_i}$$

- *Mean squared logarithmic error* - RMSLE measures the ratio between the actual and the predicted value as follows:

$$RMSLE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(log\left(y_t + 1\right) - log\left(\hat{y_t} + 1\right)\right)^2}$$

- *Log-Cosh loss function* - is the logarithm of the hyperbolic cosine of the prediction error.

$$LOG{-}COSH = \sum_{i=1}^{n} log\left(cosh\left(\hat{y}_t - y_t\right)\right)$$

For realizing the goals of our empirical work we choose to set the MSE as a standard instrument for error measuring, while training our neural networks. Thus, MSE is part of the training procedure for the Simple RNN, LSTM, GRU but also for the hybrid FNN-HAR.

After considering the various loss functions one should select the appropriate optimization algorithm, that helps for learning the optimal parameters of the corresponding model.

Optimization functions

Optimization functions help for minimizing or maximizing the loss functions. The relevant for this work optimization functions could are presented bellow.

Gradient descent is the most important technique of how the intelligent systems are trained. Typical for the algorithm are the small steps taken in direction of the negative error gradient of the parameter θ.

$$\theta = \theta - \eta \nabla Q\left(\theta\right)$$

with η being the learning rate and $\nabla Q(\theta)$ is the gradient of the loss function $Q(\theta)$ with respect to θ.

The algorithm measures the local gradient of the loss function with respect to the gradient vector θ, based on this, the gradient descent goes in the direction of descending gradient. This procedure is completed when the gradient equals zero, i.e. it already achieved the minimum. Concretely, it starts by passing random values to θ, a step called *random initialization*, in the next phase it is improved gradually, whereby the algorithm converges in direction of the minimum in small steps, while decreasing the loss function at the same time. Unfortunately, this procedure has the drawback of getting easily stuck in local minimum. Another disadvantage, associated with the traditional gradient descent, is that it calculates the gradient of the whole training data set at once, which results in a very slow process for larger data sets.

In order to overcome this problem, many of the deep learning algorithms use the *stochastic gradient descent* or *SGD*. It ensures an update in smaller steps for each training example and needs less time in general. SGD picks a random instance and estimates the gradients only

from this small sample. However, this procedure is accompanied from high variance parameter updates and unstable convergence, since the loss function bounces up and down and decreases only on average.

A solution of this problematic is the *mini batch gradient descent*. In principle, the algorithm estimates the gradients on small random samples of instances, that are called *mini batches*. Each mini batch consists of n observations, defining the *batch size*. This batch size is responsible when the neural network's weights should be updated. Afterwards, the whole gradient is calculated with the average value from all mini batches, whereby each iteration results in the following update:

$$\theta = \theta - \eta \nabla Q\left(\theta, x_i, y_i\right)$$

with x_i being m mini batches from the whole training data set.

For training our neural networks we stick to the SGD with mini batches. After some tests for the optimal batch size, we choose mini batches of size 500 for all trained neural networks.

Faster optimization algorithms

So far we have considered the classic Stochastic gradient descent with mini batches as the only solution for improving the time performance of out algorithms when larger data sets are used. In fact, this could be achieved also with some faster optimizers. We would cover few of the most important ones in this paragraph.

Momentum optimization considers at each iteration, in contrast to stochastic gradient descent, what the previous gradients were. In each step the algorithm subtracs the local gradient from the momentum vector **m** and actualize the weights just by adding this vector. So, expressed differently, at that point the gradient serves as an acceleration but not as a speed as before.

The equation of the Momentum optimization takes the form as shown below:

$$\begin{aligned} m &\leftarrow \beta - \eta \nabla Q\left(\theta\right) \\ \theta &\leftarrow \theta + m \end{aligned} \tag{56}$$

with β being a parameter preventing the momentum from growing too large. Unfortunately, adding one more parameter, that should be tuned, could be considered as a disadvantage of the algorithm.

AdaGrad decays the learning rate, however, it does it faster for steep dimensions than for dimensions with gentler slopes. This feature has the name *adaptive learning rate* and contributes for a quick convergence of the resulting updates toward the global optimum. Another advantage of the algorithm is that it does not require as much tuning of the learning rate η. The corresponding formulas are presented below.

$$\begin{aligned} s &\leftarrow s + \nabla Q(\theta) \otimes \nabla Q(\theta) \\ \theta &\leftarrow \theta - \eta Q(\theta) \oslash \sqrt{s + \epsilon} \end{aligned} \tag{57}$$

with s being the accumulated square og gradients, $\otimes$ - the symbol for element-wise multiplication and $\oslash$ represent the element-wise division.

It was, however, proven that the AdaGrad algorithm slows down too quickly and actually never converges to the optimum. Due to this negative side of the algorithm, the *RMSProp* was proposed. It accumulates only the most recent iterations using exponential decay at the very beginning. *RMSProp* is obtained by the means of the equations below.

$$\begin{aligned} s &\leftarrow \beta s + (1 - \beta) \nabla Q(\theta) \otimes \nabla Q(\theta) \\ \theta &\leftarrow \theta - \eta \nabla Q(\theta) \otimes \sqrt{s + \epsilon} \end{aligned} \tag{58}$$

A combination of RMSProp and Momentum optimization is called *Adam optimization*. As the Momentum optimization it includes the exponentially decaying average of the past gradients, and, on the other hand, as the RMSProp it involves an exponentially decaying average of past squared gradients.

$$\begin{aligned} m &\leftarrow \beta_1 m - (1 - \beta_1) \nabla Q(\theta) \\ s &\leftarrow \beta_2 s + (1 - \beta_2) \nabla Q(\theta) \otimes \nabla Q(\theta) \\ m &\leftarrow \frac{m}{1 - \beta_1^t} \\ s &\leftarrow \frac{s}{1 - \beta_2^t} \\ \theta &\leftarrow \theta + \eta m \oslash \sqrt{s + \epsilon} \end{aligned} \tag{59}$$

Similarly to RMSProp and AdaGrad the Adam optimization is an adaptive learning rate algorithm, therefore it does not need fine tuning for η, therefore the technique is mostly preferred for training neural networks. For out empirical work and learning the corresponding NNs, we make use of the Adam optimizer, since it combines the best properties of the other two adaptive algorithms. Additionally, it is easy to configure and the default configuration

parameters do well on most of the problems. After testing different learning rates, we choose to apply $\eta = 0.005$ for all of the proposed models.

Avoiding overfitting through regularization

Neural networks typically have dozens or even thousands of parameters which makes the NNs able to fit in the best way to complex data. However, such occasions often refer to a problem called overfitting, which goes hand in hand with its opposite state - underfitting. Both concepts are introduced below:

- *Overfitting* - is the state, when the model memorizes too exactly how patterns from the training data look like and fail to generalize these patters on the test data set. In other words, a large deviation of the error from the training to the test data set could be expected.
- *Underfitting* - is the opposite situation, when the model fails in learning the patters from the data and, thus, the predicting error gets larger.

As introduced above, neural networks often have to deal with the problem of overfitting. Due to this fact, researchers developed some techniques for solving the issue.

Early stopping is a simple strategy for stopping the training when its performance on the test data set start decreasing. The technique works very well in practice but it could be even improved if combined with other techniques.

Dropout is the most popular regularization technique, proposed by Krizhevsky et al. (2012). Despite its efficiency, the algorithm works pretty simply - at every training step every neuron has the probability p to be excluded or "dropped out" from the network, i.e. it will be completely ignored in this step but probably included in the next one. This parameter p is called the *dropout rate* and is usually defined to be 0.5.

Max-norm regularization is another relatively popular technique for avoiding overfitting. It sets an upper bound on the magnitude of the weight vector for every neuron, whereas it applies the projected gradient descent to enforce the constraint. The main advantage of it is that the network cannot "explode" even when larger learning rates are included.

However, we aim a possibly best predicting algorithms and take into account that prices of all cryptocurrencies change very dynamically, so generalizations outside the train data set will be always a challenge, therefore we avoid adding a new regulating layer to our neural networks.

Activation function

As we have seen in Section 4 the activation functions play probably the main role for training our nonlinear models. Essentially, they decide if a neuron should be activated or not, meaning they are responsible for deciding if the given information is relevant or not. Referring to Equation (35), the activation function is the one "firing" or "not firing" the corresponding hidden unit h_i. This arise the question if we could train a neural network without having specific activations functions. The answer is yes, however, in such a case our models would be nothing more than a simple regression. We discuss below the most popular functions contributing for the aimed nonlinearity of our neural networks:

- *Sigmoid* - is probably the most popular nonlinear activation function. Reason for this is that it gives solution to the already discussed gradient vanishing problem. However, the function also has its negative side - its outputs are non-zero centered, which causes a very slow convergence. Additionally, the function saturates and kills gradients. Because of its form, presented below, the function takes values in range between 0 and 1.

$$f(x) = \frac{1}{1 + e^{-x}}$$

- *Tanh* - has very similar characteristics to the *Sigmoid* function, however, the gradient is stronger for *Tanh*, i.e. its derivatives are slightly steeper. Its bounds are -1 and 1,

$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$

- *ReLU* - or *Rectified linear units* works very well in practice, plus, it is very fast compared to other activation functions. Most importantly, it does not have upper bound which reduces some issues connected with the gradient descent. Additionally, the sparsity of the activation could be pointed out as another advantage - in a large complex neural network only around 50% of the neurons are "fired" due to characteristics of *ReLU*, this leads to a lighter version of the model. However, this could be detected also as a drawback for some cases. The so called *dying ReLU* situation occurs when some neurons "die out", i.e. they start outputting only zeros. When this happen, it is very unlikely that these particular units come back to life.

$$f(x) = max(0, x)$$

- *leaky ReLU* - as a solution for the above mentioned problems of textitReLU the *leaky ReLU*. It has an additional parameter α defining how much from the given information should "leak". In other words, this hyperparameter ensures that these around 50% of the neurons never "die out". The mathematical equation hiding behind *leaky ReLU* looks like:

$$f(x) = max(\alpha x, x)$$

  where α is typically set to 0.01.

- *ELU* - or *Exponential linear unit* was proposed by Clevert et al. (2015). The authors demonstrated that *ELU* outperforms all other suggested nonlinear activation functions in the test data set. The function obtains the following form:

$$f(x) = \begin{cases} \alpha(exp(x) - 1) & x < 0 \\ x & x \geq 0 \end{cases}$$

  which allows it to take negative values if $x < 0$. The form of ELU avoids the dyings neurons issue and also contributes for an average output closer to zero. The parameter α signalizes the value that the function approaches for x being relative large and negative. Furthermore, the function is smooth. However, *ELU* is also characterized by some disadvantages, e.g. it is slower in computation.

After discussing all the main activation functions it is natural for us to ask which function suits best for solving our regression problem. Since we deal with predicting the realized volatility, we need an activation function that gives values in range between -1 and 1, has an average around zero and, additionally, works fine with large amount of data. All of these criterions are fulfilled by ELU, so we apply it as the activation function in all layers of our neural networks.

Number of units and layers

We already introduced the concept of overfitting which is caused by the larger number of parameters, that the neural networks have. Because of that property NNs are able to predict with sufficiently good accuracy outputs from data sets with more complex data patterns. However, since we want to compare the outcomes from different linear and nonlinear models

with different levels of memorizing information, we want to keep our NNs as simple as possible. In other words, doing this, we ensure a fair comparison between the linear HAR-RV and the RNNs.

Because of the stated above reason, all applied neural networks - SRN, LSTM, GRU but also the hybrid FNN-HAR consist just from three layers (an input layer, a hidden layer and an output layer), whereby the number of hidden units is only two. This results in a neural network constructed by only 11 parameters.

Number of epochs

An epoch is another important hyperparameter in training neural networks. One epoch results, when the entire data set is once passed forward and then once backward through the whole neural network.

Since we apply an iterative process like gradient descent, we should consider that updating the neural network's weight with just one pass or epoch would result in underfitting, and a large number of epochs would lead most likely to overfitting. So what is the right number of passing the data forward and backward through the neural network?

Unfortunately, there is no clear rule how to define the most suitable number of epochs. After some trails we detected that 100 epochs fit well to out data and does not cause neither underfitting nor overfitting.

Summary

In summary, the training and fine tuning of neural networks is as complex process consisting of many small steps. However, in order to ensure a competitive for our linear and nonlinear models environment for predicting the realized volatility of cryptocurrencies, we keep the applied neural networks as simple as possible. Due to this fact, the NNs are constructed by only 3 layers with 2 hidden units, whereby the neurons are "fired" by the ELU activation function. We make use of the *Stochastic gradient descent* with Adam as an adaptive optimizer, whereby the starting *learning rate* or η is set to 0.005. Each batch covers 500 observations and the number of forward and backward passes or the so called *number of epochs* is equal to 100. These parameters are summarized in Table (1).

Since we use the HAR-RV as a benchmark model, a comparable conditions for the neural networks should be provided. Equation (55) uses as explanatory factors only variables from one time period, therefore in the neural networks as inputs we choose only the daily realized volatility from the current time period t for out attempt to forecast its value in $t + 24h$, i.e.

Paramater	Value
Number of hidden layers	1
Number of hidden units	2
Activation function	ELU
Optimizer	Adam
Learning rate	0.005
Batch size	500
Epochs	100
Total number of parameters	11

Table 1: Hyperparameters for the recurrent neural networks (SRN, LSTM, GRU).

 RVOLrnn

the target value corresponds to the same variable with a shift from 24 hours.

All parameters, stated above, refer to all recurrent neural networks proposed in our study - SRN, LSTM and GRU. The exact training of the Networks could be found in the *Quantlet*, that is cited in the caption of Table (1). Additionally, the hybrid model between FNN and HAR-RV, also makes use of these parameters. However, it requires a more detailed discussion, therefore we regard it separately in the next section.

5.3 Training the hybrid FNN-HAR

In Section 4.3 we introduced the hybrid model, which aims to combine the memory feature of the HAR-RV model with the nonlinearity of the classic feedforward neural network. In order to achieve these goals, we have to calculate the daily realized volatility - $RV_t^{(1d)}$, the weekly one - $RV_t^{(7d)}$ and the monthly one - $RV_t^{(30d)}$ from Equation (55). These three variables are then used as input variables for the feed forward neural network, whereby we consider the values only from the time period t for forecasting the realized volatility in $t + 24h$ of our moving 24h-window, meaning that out target value equals the $RV_{t+24h}^{(1d)}$. So, at the end, we end up with a model taking over the memory of HAR-RV but also using the nonlinearity of a simple FNN.

As Figure (7) illustrates, the hybrid model FNN-HAR assumes only two neurons and just one hidden layer, whereby the input variables are the daily, weekly and monthly realized volatiles, that are calculated after their adjusted formulas from Section 5.1. All other hyper parameters, associated with NNs, remain unchanged and take the values from the RNNs

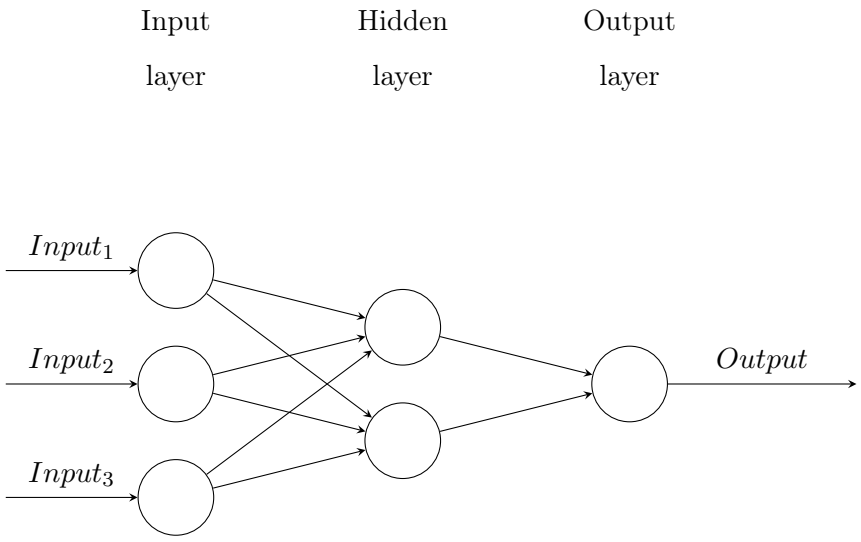

Figure 7: Structure of the hybrid FNN-HAR model.

RVOLharfnn

shown in Table 1. The corresponding *Quantlet* for building the hybrid model stays in the caption of Figure (7).

6 Data

For the realization of the goals of this particular master thesis we use the data, which was granted by Bruno Spilak from Dyos. At that point, we want to thank him for sharing the high frequency data of 6 cryptocurrencies with us.

The offered data covers the time period from 2016/01/01 to 2018/08/31 and includes 5 minutes frequency data of Bitcoin (btc), Ethereum (eth), Litecoin (ltc), StarCoin (str), Monero (xmr) and Ripple (xrp). The 5 minutes intervals data set reveals information about the opening, closing, highest and lowest prices in USD for all six available cryptocurrencies plus the corresponding volume traded (in units). The time series of the daily closing prices at 00:00:00 are presented at Figure (8).

From Figure (8) it becomes obvious that some of the coins emerge on the markets in later periods, therefore we choose to concentrate on the last 365 days from the data, namely from 2017/08/31 to 2018/08/31. The corresponding descriptive statistics for the six coins are represented in Table (2).

For the purpose of our particular work creating of a portfolio from all available cryptocurrencies would provide more suitable for interpretation results and, additionally, would show us broader perspectives for financial applications of the forecasted realized volatility. Therefore,

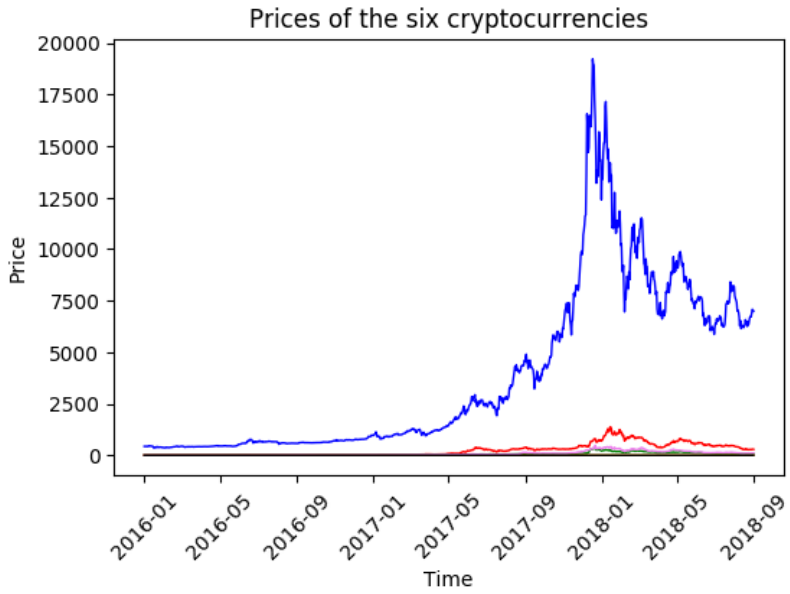

Figure 8: Princes of all coins - btc (blue), eth (red), ltc (green), str (orange), xmr (violet), xrp (black).

RVOLprep

	btc	eth	ltc	str	xmr	xrp
Min	2.93×10^3	2.00×10^2	3.35×10^1	6.47×10^{-3}	7.22×10^1	1.47×10^{-1}
Max	1.99×10^4	1.42×10^3	3.70×10^2	9.18×10^{-1}	4.74×10^2	3.28
Mean	8.35×10^3	5.53×10^2	1.24×10^2	2.37×10^{-1}	1.89×10^2	6.29×10^{-1}
Median	7.66×10^3	4.75×10^2	1.13×10^2	2.29×10^{-1}	1.65×10^2	4.99×10^{-1}
Std	3.10×10^3	2.48×10^2	6.62×10^1	1.65×10^{-1}	9.04×10^1	4.74×10^{-1}
1st Quartile	6.43×10^3	3.32×10^2	6.73×10^1	8.18×10^{-2}	1.15×10^2	2.52×10^{-1}
3rd Quartile	9.52×10^3	7.03×10^2	1.61×10^2	3.29×10^{-1}	2.52×10^2	8.13×10^{-1}

Table 2: Some descriptive statistics of the prices of all six cryptocurrencies.

RVOLprep

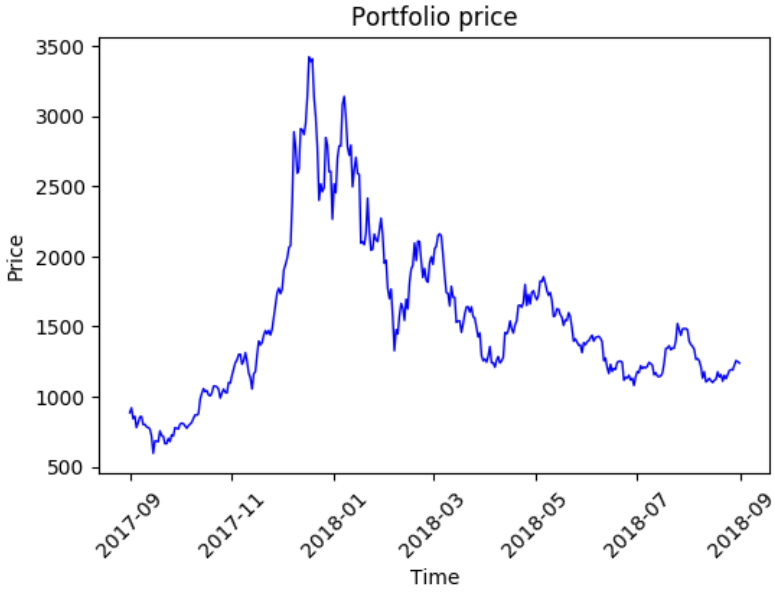

Figure 9: Price evolution of a price weighted portfolio from six coins.

RVOLprep

we combine all coins in a single portfolio. Unfortunately, we lack of market capitalization data required to follow the way the CRIX index was established by Trimborn and Härdle (2016), so we assume the strategy of a price weighted portfolio. Its price evolution follows the formula as given below:

$$P_{portf,t} = \omega_{btc}P_{btc,t} + \omega_{eth}P_{eth,t} + \omega_{ltc}P_{ltc,t} + \omega_{str}P_{str,t} + \omega_{xmr}P_{xmr,t} + \omega_{xrp}P_{xrp,t} \tag{60}$$

with ω and P being respectively the corresponding coin's weight and price. Additionally, $\omega_{btc} + \omega_{eth} + \omega_{ltc} + \omega_{str} + \omega_{xmr} + \omega_{xrp} = 1$, so in an equally weighted portfolio all single omegas obtain the value 1/6.

The evolution of the daily closing prices (at 00:00:00) of our portfolio for this certain period of time is visualized in Figure (9).

Because of the fact that our empirical work relates to the daily realized volatility, we have to calculate the log returns of our portfolio according to the equation below.

$$R_t = ln\left(\frac{P_{t+1}}{P_t}\right) \tag{61}$$

whereby the log returns of the portfolio are presented in Figure (10), and some descriptive statistics like the mean, standard deviation, skewness, kurtosis and the result from a normality test could be found in Table (3):

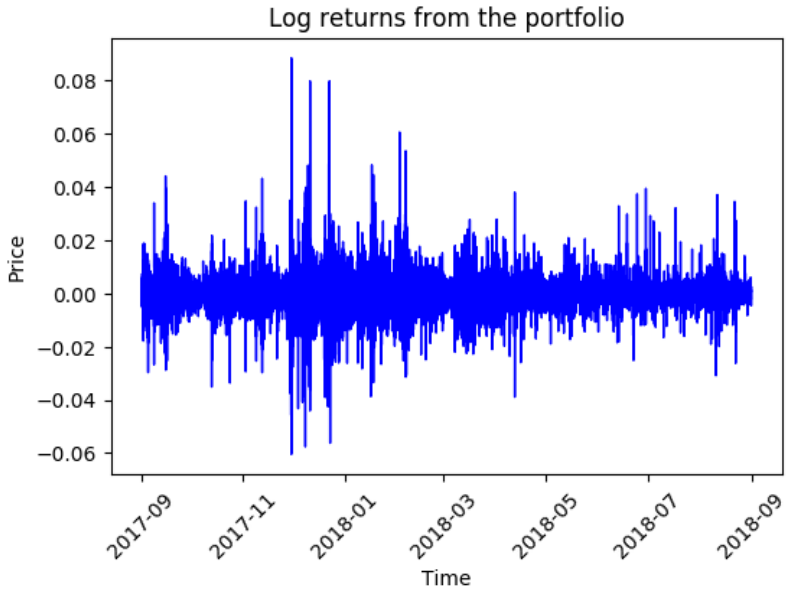

Figure 10: Log returns of the portfolio.

RVOLprep

	btc	eth	ltc	str	xmr	xrp	portfolio
Mean	3.80×10^{-6}	-3.01×10^{-6}	-1.51×10^{-6}	2.18×10^{-5}	-1.80×10^{-6}	2.45×10^{-6}	3.27×10^{-6}
Std .	3.79×10^{-3}	4.19×10^{-3}	4.98×10^{-3}	7.65×10^{-3}	5.18×10^{-3}	5.93×10^{-3}	3.71×10^{-3}
Skew.	2.46×10^{-1}	3.16×10^{-1}	4.41×10^{-1}	2.69×10^{-1}	5.72×10^{-2}	2.76×10^{-1}	2.71×10^{-1}
Kurt.	2.45×10^{1}	2.14×10^{1}	2.28×10^{1}	1.59×10^{1}	1.10×10^{1}	2.68×10^{1}	2.59×10^{1}
JB	2.63×10^{6}	2.00×10^{6}	2.28×10^{6}	1.11×10^{6}	5.27×10^{5}	3.15×10^{6}	2.93×10^{6}

Table 3: Some descriptive statistics of the log returns.

RVOLprep

Unfortunately, the data available for all the six coins is not enough for a good training of neural networks, therefore we decided to work with rolling windows, i.e. we consider a day from 24 hours (288 observations with 5 minutes intervals), so to each 24 hours we keep adding the next 5 minutes, while at the same time removing the first observation from this 288 rows sequence. In other words, when we forecast the future realized volatility, we consider its value in 24 hours from period t, i.e. in $t + 24h$. In order not to lose any information from our relatively short sequence of data, all values (log returns, daily realized volatility, weekly realized volatility and monthly realized volatility) are firstly estimated for the whole data set from 2017/08/31 to 2018/08/31. This is done due to the fact that if concentrating only on specific periods from the given time series, we would lose the information from the first 30 days in order to find the monthly realized volatility. After calculating all of our necessary variables, we could move forward to the exact empirical results from our research.

7 Empirical application

Cryptocurrencies are known for their price fluctuations, which makes the accurate prediction of volatility extremely difficult.

As one can notice from Figure (10) even for our short data set there are periods of time, that distinguish themselves with high volatility in the log returns, and, on the other hand, the last months seem to be relatively quiet, i.e. there are no huge price deviations. Capturing these facts, it is almost natural to challenge the accuracy of our models in high volatility times and also in periods with relatively low volatility.

Additionally, we vary the amount of data available for training and test data sets. This is done in order to check the ability of the models to learn in shorter time intervals and also to control for their predicting power for different time horizons.

The predicting power of all applied models is judged by the means of some classic coefficients for estimating the accuracy of statistic forecasts or the so called loss functions from Section 5.2.

Our models are also compared to each other. For this reason we apply the standard for such occasions *Diebold Mariano test (DM test)*, named after its authors (Diebold and Mariano (2002)). Essentially, the test compares the predicting power of a couple of models with forecast errors respectively $e_{i1}, \ldots, e_{it}$ and $e_{j1}, \ldots, e_{jt}$, whereby the quality of each forecast is estimated by some loss function. The null hypothesis for the test signalizes equal predictive accuracy of the models for all time periods. The idea behind the test corresponds to the following

equations:

$$E\left(d_t\right) = 0$$

with

$$d_t = g\left(e_{it}\right) - g\left(e_{jt}\right)$$

where g stays for the corresponding loss function.

All codes, that are used for the visualization of the forecasts and calculating the Diebold Mariano tests, are part of the cited *Quantlet* below Figure (11) and Figure (12).

Our empirical work is finalized by a financial application of the models. The predicted realized volatility is used as a proxy for the true volatility in calculating the Value at Risk (VaR) for multiple periods ahead. The strength of this approach is controlled by some backtests.

Before moving to the exact empirical results, let we firstly introduce the *Naïve approach*, that is frequently used as a benchmark when working with time series. The Naïve approach is one of the most straightforward and simplest models for making forecasts with time series due to the fact that it uses as forecasts the true values from the previous period. In other words, when predicting the realized volatility in $t + 24h$, the Naïve approach would just assume that the true values for daily realized volatility in period t are still accurate in $t + 24h$. This kind of making predictions is considered to work quite well for many economical and financial time series, therefore it is usually assumed as a benchmark in many scientific works. However, according to Zakamulin (2014) the Naïve approach cannot be used for making forecasts in the long run. It is also criticized because of being too persistent. Nevertheless, the model is used as a benchmark in our empirical work.

7.1 Accuracy of the predicting models

In order to look how accurately the models work in various situations and what predicting errors they demonstrate, we choose to concentrate on two separate time intervals from our data set. The first one covers the last three months (2017/10/01 - 2017/12/31) of 2017, that we characterized with higher price fluctuations, and the second time window consists of the observations from 2018/04/15 to 2018/07/15.

7.1.1 High frequency times

Concentrating on the high volatility times, we consider two different cases. Firstly, we take the whole period from 2017/10/01 to 2017/12/31, which is separated after the standard procedure

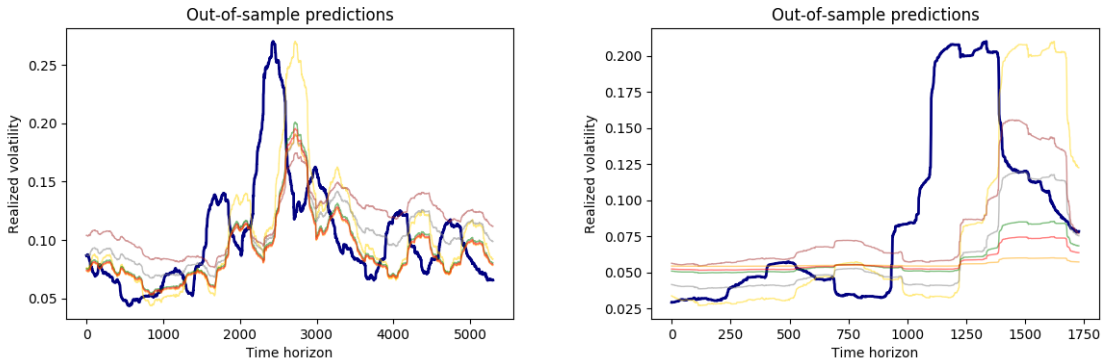

Figure 11: Predicted realized volatility for high volatility times - target value (navy), Naïve (yellow), HAR-RV (grey), FNN-HAR (brown), SRN (green), LSTM (orange), GRU (red). Left panel - long case, right panel - short case.

RVOLaccur

	Naïve	HAR	FNN-HAR	SRN	LSTM	GRU
RMSE	5.40	2.92	2.96	3.00	2.97	2.98
MAE	3.69	1.97	2.05	2.21	2.12	2.16
RMSLE	2.39×10^{-1}	7.01×10^{-2}	7.27×10^{-2}	7.46×10^{-2}	7.28×10^{-2}	7.36×10^{-2}
RMSE	4.93	4.44	4.87	4.42	4.45	4.44
MAE	3.60	3.25	3.81	2.99	2.93	2.95
RMSLE	1.83×10^{-1}	1.51×10^{-1}	1.84×10^{-1}	1.49×10^{-1}	1.51×10^{-1}	1.50×10^{-1}

Table 4: Prediction errors for the high volatility times (long). Upper panel - training data set predictions errors, bottom panel - out-of-sample prediction errors. Note that because of the low numbers all numbers are multiplied by 10^2.

RVOLaccur

from data science, namely the first 80% belong to the training data set and the rest 20% to the test data set. Doing this, we maintain the sequence of the time series and their time dependence. Afterwards, the training data set is reduced by extracting only the middle 30 days from the time interval and the predicting models are trained once again. For simplicity, until the end of the paper we call these both cases of longer and shorter training data sets respectively *long case* and *short case*.

The forecasted realized volatilities from our predicting models are shown in Figure (11), where one could easily notice that all of the outputs from our models have very similar course of the predicted time series. They follow almost exactly the evolution of the target values, however, with the only difference that all of the forecasts seem to be unable to reproduce such

	Naïve	HAR	FNN-HAR	SRN	LSTM	GRU
RMSE	5.03	2.17	2.48	2.28	2.27	2.37
MAE	3.33	1.65	1.84	1.71	1.71	1.78
RMSLE	2.11×10^{-1}	4.06×10^{-2}	5.34×10^{-2}	4.47×10^{-2}	4.44×10^{-2}	4.86×10^{-2}
RMSE	7.15	6.27	6.58	6.17	6.17	6.37
MAE	5.08	3.62	4.25	3.72	3.70	4.11
RMSLE	4.02×10^{-1}	3.13×10^{-1}	3.46×10^{-1}	3.02×10^{-1}	3.02×10^{-1}	3.22×10^{-1}

Table 5: Prediction errors for the high volatility times (short). Upper panel - training data set predictions errors, bottom panel - out-of-sample prediction errors. Note that because of the low numbers all numbers are multiplied by 10^2.

RVOLaccur

	Naïve	HAR	FNN-HAR	SRN	LSTM
HAR	-9.83×10^{-1}				
FNN-HAR	-1.01×10^{-1}	1.97**			
SRN	-1.08	-8.76×10^{-2}	-1.15		
LSTM	-9.29×10^{-1}	5.63×10^{-2}	-9.92×10^{-1}	4.90×10^{-1}	
GRU	-9.81×10^{-1}	1.79×10^{-2}	-1.04	5.29×10^{-1}	-4.12×10^{-1}
HAR	-9.60×10^{-1}				
FNN-HAR	-9.73×10^{-1}	7.31×10^{-1}			
SRN	-1.19	-6.52×10^{-1}	-1.00		
LSTM	-1.18	-7.28×10^{-1}	-9.88×10^{-1}	-3.89×10^{-1}	
GRU	-1.22	3.41×10^{-1}	-1.02	9.08×10^{-1}	8.80×10^{-1}

* $p < 0.10$, ** $p < 0.05$, *** $p < 0.01$

Table 6: Test statistic from the Diebold Mariano test for the out-of-sample predictions for high volatility times. Upper panel - long case, bottom panel - short case.

RVOLaccur

extreme values.

The errors from the forecasts are summarized in Table (4) and Table (5) and the corresponding results from the *Diebold Mariano test* for the out-of-sample predictions are covered in Table (6). What occurs to be interesting, is that for the *long case* with high volatility time series all of our predictions seem to have relatively similar results, this fact is supported also by the results of the DM test, which does no find any model to be significantly different than the others, only exception makes the couple of HAR and FNN-HAR, which rather signalizes that the hybrid model has the worst predicting power for this scenario. On the other hand, looking at the *short case*, one could notice that the recurrent neural networks tend to perform slightly better than the other models, however, the difference in predicting accuracy is not significant.

At that point of our research it is important to be stressed that we deal with financial time series from cryptocurrencies, which suggests significant jumps and noise in the data. It is natural that such patters in the data sets cannot be captured by linear models and would rather require nonlinear models for more accurate predictions. In such a case, why do not our hybrid model and the recurrent neural networks significantly outperform the linear HAR model? We assign this to the fact that in our research these nonlinear algorithms are kept very simple, i.e. they do not have the command over many hyperparameters like hidden layers and units, thus they become unable to reveal their potential for predicting in situations with high volatility.

7.1.2 Low frequency times

In this section we regard the ability of the models to forecast the realized volatility, when they are trained in low volatility times. The procedure from Section 7.1.1 is repeated, i.e. the length of the training data set is varied as before, so we end up having both situations *long case* and *short case* again.

The forecasts from our algorithms are plotted in Figure (12), whereby the corresponding prediction errors for the long case and these for the short case are summarized respectively in Table (7) and Table (8).

As one could notice from the graph, all of our models are able to capture the fluctuation and the direction of the target values and, as founded out in the previous section, none of the models reproduces such extreme values except the Naïve model, which follows strictly the true values from the previous 24 hours. However, one more thing could be distinguished as

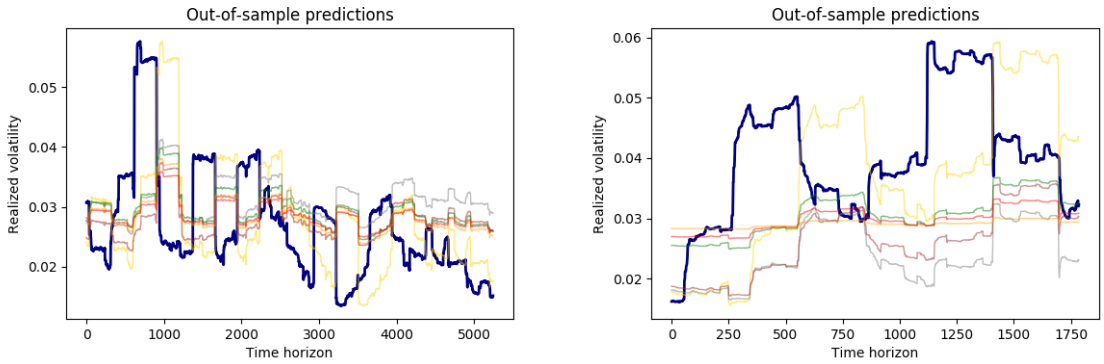

Figure 12: Predicted realized volatility for low volatility times - target value (navy), Naïve (yellow), HAR-RV (grey), FNN-HAR (brown), SRN (green), LSTM (orange), GRU (red). Left panel - long case, right panel - short case.

RVOLaccur

	Naïve	HAR	FNN-HAR	SRN	LSTM	GRU
RMSE	1.41	9.10×10^{-1}	9.83×10^{-1}	9.62×10^{-1}	1.00	1.03
MAE	1.09	7.16×10^{-1}	7.32×10^{-1}	7.07×10^{-1}	7.26×10^{-1}	7.51×10^{-1}
RMSLE	1.85×10^{-2}	7.69×10^{-3}	8.95×10^{-3}	8.56×10^{-3}	9.28×10^{-3}	9.77×10^{-3}
RMSE	1.28	1.14	9.60×10^{-1}	1.01	9.81×10^{-1}	9.67×10^{-1}
MAE	1.05	9.88×10^{-1}	8.01×10^{-1}	8.63×10^{-1}	8.25×10^{-1}	8.05×10^{-1}
RMSLE	1.52×10^{-2}	1.22×10^{-2}	8.66×10^{-3}	9.53×10^{-3}	9.03×10^{-3}	8.77×10^{-3}

Table 7: Prediction errors for the low volatility times (long). Upper panel - training data set predictions errors, bottom panel - out-of-sample prediction errors. Note that because of the low numbers all numbers are multiplied by 10^2.

RVOLaccur

well - the LSTM model seems to have much smaller variations of its predictions, especially for the case with less amount of training data.

The corresponding Table (7) for errors from the predictions in the *long case* demonstrated that the nonlinear models tend to have smaller errors when compared to the other models. The statement is also confirmed by the *p values* from the upper panel of Table (9), where the null hypothesis for equality in the predicting models is rejected always if one of the models in the couple is the Naïve or the HAR-RV model. This fact and also the results from Table (9) are a sign that the nonlinear models are stronger for a longer training period during the low volatility times.

The whole picture does not change significantly for the *short case*. Judging by the results

	Naïve	HAR	FNN-HAR	SRN	LSTM	GRU
RMSE	1.48	6.56×10^{-1}	7.72×10^{-1}	7.21×10^{-1}	7.21×10^{-1}	7.97×10^{-1}
MAE	1.13	5.15×10^{-1}	5.80×10^{-1}	5.25×10^{-1}	5.26×10^{-1}	5.95×10^{-1}
RMSLE	2.04×10^{-2}	4.03×10^{-3}	5.56×10^{-3}	4.86×10^{-3}	4.85×10^{-3}	5.93×10^{-3}
RMSE	1.54	1.96	1.68	1.36	1.35	1.41
MAE	1.38	1.62	1.39	1.02	1.01	1.09
RMSLE	2.19×10^{-2}	3.59×10^{-2}	2.62×10^{-2}	1.71×10^{-2}	1.70×10^{-2}	1.83×10^{-2}

Table 8: Prediction errors for the low volatility times (short). Upper panel - training data set predictions errors, bottom panel - out-of-sample prediction errors. Note that because of the low numbers all numbers are multiplied by 10^2.

RVOLaccur

	Naïve	HAR	FNN-HAR	SRN	LSTM
HAR	-7.97×10^{-1}				
FNN-HAR	−1.73***	−4.55**			
SRN	−1.69***	−3.36**	1.95**		
LSTM	−1.82***	−3.06**	6.74×10^{-1}	−1.18	
GRU	−1.78***	−3.53**	2.65×10^{-1}	−1.48	−1.11
HAR	1.04				
FNN-HAR	4.76×10^{-1}	−2.09***			
SRN	-8.51×10^{-1}	−2.44***	−2.53**		
LSTM	-8.78×10^{-1}	−2.44***	−2.53**	−2.12**	
GRU	-5.08×10^{-1}	−2.67***	−3.24**	6.24×10^{-1}	6.74×10^{-1}

* $p < 0.10$, ** $p < 0.05$, *** $p < 0.01$

Table 9: Test statistic from the Diebold Mariano test for the out-of-sample predictions for low volatility time. Upper panel - long case, bottom panel - short case.

RVOLaccur

in Table (8), one could say that the nonlinear models are still better than the Naïve approach and the HAR-RV, however, there is a new finding here. All of our recurrent neural networks seem to have outperformed the other algorithms. Looking at the outputs from bottom panel of Table (9), it becomes obvious that particularly the simple recurrent neural network and the LSTM models have different forecasting results from the other models.

In summary, one could say that the nonlinear models perform significantly better in both - the *long case* and *short case*. The finding from the high volatility times scenario that the recurrent neural networks tend to predict with lower error remains valid also for the low volatility times, additionally, it is shown that the accuracy of SRN and LSTM is significantly different than that of the other models. This fact would suggest that probably the recurrent neural networks and concretely the SRN and LSTM are better predicting models for low volatility times. Such a finding also corresponds to our hypothesis that the low number of hyperparametrs in the used nonlinear models is the reason for their inability to perform significantly better than the Naïve and HAR models during high volatility times. However, for simpler data like this from the low volatility times it turns out that, even restricted to only 11 hyperparameteres, the nonlinear models are better in making forecasts and also required less information in the training phase.

7.2 Financial application

Once having the forecasts for the realized volatility in the next 24h for high and low volatility times, it will be useful if we could find a financial application for our outputs.

Degiannakis and Potamia (2017) checked the ability of intra-day returns to predict correctly the *Value at Risk (VaR)* and the *Expected shortfall (ES)* for stock indices, commodities and foreign exchange rates. However, according to the provided results this approach does not provide any good results for 10 steps ahead, 20 steps ahead and multiple periods ahead VaR and ES at confidence level of 95%, 97.5% and 99%. Giot and Laurent (2004) also contributed to this area of research. In this particular paper, the VaR based on the predictions for realized volatility, was calculated for one day ahead through moving windows, however, according to the authors this approach does not seem to be an improvement of the standard methods. On the other hand, Kruse (2006) showed that models linked to the realized volatility are able to build suitable VaR models for S&P 500 if it comes to one period ahead forecasts. The author also noted that the performance of models, which assume normality, is relatively good. Additionally, Bedowska-Sojka (2015) also proved that VaR estimates, obtained from

the classic models for daily returns, have comparable results to those calculated with realized volatilities. However, the long memory feature and the asymmetry of realized volatility seem to have improved slightly the predictions.

Furthermore, realized volatility is also used for *options pricing*. So, for example, Corsi et al. (2013) presented that the proposed RV model outperforms the competing stochastic volatility option pricing models for S&P's 500 index options. Stentoft (2008) also demonstrated that modeling the realized volatility and using it for option pricing leads to better results than the traditional models based on inter-daily data.

Considering all of these stated facts, we chose to apply our forecasts for calculating the multiple periods ahead VaR, which validity we control with the classic backtesting methods. Below are described the VaR method and the backtests we used. The *Quantlet* containing the codes for this financial application is cited in the captions of Table (10) and (11).

7.2.1 Value at Risk

Value at Risk quantifies the market risk of a portfolio to market fluctuations in the next periods. Usually, this number is defined as the amount of money, that could be lost within a certain horizon, whereby the probability for this event is equal to α.

$$P\left(P_t \leq P_{t-1} - VaR_t\left(\alpha\right)\right) = \alpha$$

and if it comes to log returns, the VaR is usually defined as follows:

$$P\left(r_t \leq -VaR_t\left(\alpha\right)\right) = \alpha$$

In the rest of our empirical work VaR refers to the left tail of our portfolio returns distribution (long position). In other words, it takes the following form:

$$P\left(r_t \leq -VaR_t^l\left(\alpha\right)|\Omega_{t-1}\right) = \alpha$$

7.2.2 Backtesting

Below we regard two of the methods for VaR backtesting, which are applied in our research.

Kupiec (1995) proposed a test called the proportion of failures (POF) coverage test, however, mostly it is cited in the literature just like the *Kupiec's test* or *Unconditional coverage*

test. The proposed test works with the binomial distribution approach and a likelihood ratio in order to test if the probability of exceptions corresponds to the probability p expected for a given confidence level of VaR. The null hypothesis, that our model is able for predicting VaR, is rejected if in the given data the probability of exceptions is different than p. The test statistic takes the following form:

$$LR_{UC} = -2ln\left(\frac{(1-p)^{T-x}\,p^{x}}{\left[1-\left(\frac{x}{T}\right)\right]^{T-x}\left(\frac{x}{T}\right)^{x}}\right)$$

with x being the number of failures in predictions, N - number of observations and $p = 1-\alpha$.

The next test, which is used for validating the power of our VaR predictions, is introduced by Christoffersen (1998). It measures if the probability of observing an exception on a particular day is dependent on whether an exception occurred in the previous period. The test considers only the dependency between consecutive days only, therefore it is also known as *Conditional coverage test*. Its test statistic is given as:

$$LR_{CC} = -2log\left(\frac{(1-\pi)^{n00+n10}\,\pi^{n01+n11}}{(1-\pi_o)^{n00}\,\pi_0^{n01}\,(1-\pi_1)^{n10}\,\pi_1^{n11}}\right)$$

where:

n00 - number of periods with no failures followed by a period with no failure

n10 - number of periods with failures followed by a period with no failure

n01 - number of periods with no failures followed by a period with a failure

n11 - number of periods with failures followed by a period with a failure

π_0 - probability of having a failure in t, given that there was no failure in period $t-1$

π_1 -probability of having a failure in t, given that there was a failure in period $t-1$

π - probability of having a failure in t.

7.2.3 Applicaiton of Value at Risk and backtesting

As shown in the introduction of Section 7.2, VaR could be estimated for different periods ahead. However, training 5 models for small number of steps ahead in a moving window is a very time-consuming process, which is especially costly given the parameters of our computer, therefore we decided to look how our VaR forecasts behave for multiple periods ahead (2500 steps). In other words, we consider the forecasts from all models from the *long cases* for our high and low volatility scenarios and calculate the corresponding Value at Risk for $\alpha = 0.10$ and $\alpha = 0.05$, whereby we assume normal distribution. To measure the performance of all

single models, all forecasted values of Value at Risk are compared to the real daily log returns. Since we are focused on moving windows with length of 24h and single steps from 5 minutes, this means that in the first step also daily log returns from t to $t + 24h$ should be calculated.

The results from the tests for high volatility times are summarized in Table (10) and the corresponding figures are shown in Appendix. Since the null hypotheses of the Unconditional coverage test (Kupiec's test) and the Conditional coverage test (Christoffersen's test) are rejected, one could conclude that none of our predicting models gives outputs, that are able to be used for forecasting VaR for multiple periods ahead. However, it should be taken into account that the recurrent neural networks tend to have closer number of violations to the expected exceed than the other models. This statement also corresponds to the forecasting results shown in Section (7.1).

Table (11) contains the results from the backtesting for low volatility times and the resulting visualizations are attached in Appendix. The values from Table (11) also confirm the stated above conclusion about the power of our models to predict correctly the Value at Risk. Additionally, the tendency for better forecasts from the recurrent neural networks according to the number of violations remains unchanged. However, there is a small improvement in the *p values* for the Unconditional coverage test but the null hypotheses is still rejected.

Finally, we could conclude that the predicted with our models values for realized volatility cannot be used for correctly forecasting the Value at Risk for multiple periods ahead according to the test results from the unconditional and conditional coverage tests. This could be explained with fluctuations in the prices, that are typical for cryptocurrencies, which suggests some noise and jumps in the time series. According to Corsi (2009) this fact leads to a violation in the assumptions for convergence in probability of the realized volatility to the true volatility, thus VaR would be probably estimated falsely. Since it is assumed that the realized volatility converges to the integrated volatility only in case of sufficiently high frequency, this could be pointed out as another reason for the poor performance of our models. Our demonstrated results, however, correspond to most of the empirical works in this research area, since it is still not shown that realized volatility is a clear improvement when one tries to approximate the true volatility in financial time series.

8 Conclusion

In this paper we consider the accuracy of various models for forecasting the realized volatility of cryptocurrencies in the next period. We make use of the famous heterogenous autoregressive

	Exp. exceed	Violations	UC	CC
Naïve	250	40	0.00	0.00
HAR	250	90	0.00	0.00
FNN-HAR	250	94	0.00	0.00
SRN	250	103	0.00	0.00
LSTM	250	115	0.00	0.00
GRU	250	115	0.00	0.00
Naïve	125	2	0.00	0.00
HAR	125	12	0.00	0.00
FNN-HAR	125	16	0.00	0.00
SRN	125	14	0.00	0.00
LSTM	125	19	0.00	0.00
GRU	125	19	0.00	0.00

Table 10: Backtesting for high volatility times. UC - Unconditional coverage test, CC - Conditional coverage test. Upper panel: $\alpha = 0.1$, bottom panel: $\alpha = 0.05$.

RVOLbackt

model for realized volatility (HAR-RV), which distinguishes itself with proven long memory feature, however, the model takes the form of a simple linear regression. In order to include nonlinearity in our forecasts, some techniques from machine learning techniques are applied. Since we deal with predicting concrete values for the next period of time, we involve the simple recurrent neural network (SRN) and its variations like the long short-term memory (LSTM) and the gated recurrent unit (GRU), that are trained in correspondence to their so called regression problem. In this particular paper we also proposed a hybrid model between a simple feedforward neural network and the heterogenous autoregressive model. The combination called (FNN-HAR) of these two models should have the memory feature, linked with the HAR-RV, but also the nonlinear property due to the used activation function in the neural networks. The empirical contributions from our work to this field of research are briefly summarized below.

The forecasting performance of different models was compared in order to indicate the best model for predicting the realized volatility. We trained the models in two different periods of time, which we chose according to the volatility of the log returns of our portfolio of cryptocurrencies. We defined as high volatility times the period between 2017/10/01 and 2018/01/01

	Exp. exceed	Violations	UC	CC
Naïve	250	95	0.00	0.00
HAR	250	124	0.00	0.00
FNN-HAR	250	134	0.00	0.00
SRN	250	117	0.00	0.00
LSTM	250	167	4.80×10^{-9}	0.00
GRU	250	167	4.80×10^{-9}	0.00
Naïve	125	16	0.00	0.00
HAR	125	15	0.00	0.00
FNN-HAR	125	11	0.00	0.00
SRN	125	11	0.00	0.00
LSTM	125	17	0.00	0.00
GRU	125	17	0.00	0.00

Table 11: Backtesting for low volatility times. UC - Unconditional coverage test, CC - Conditional coverage test. Upper panel: $\alpha = 0.1$, bottom panel: $\alpha = 0.05$.

RVOLbackt

and as a low volatility period the time window between 2018/04/15 and 2018/07/15. Additionally, we also regarded two cases or the so called by us *long case* and *short case*, where we reduced the amount of observations in the training data set. These variations aimed to give us more information about the ability of the algorithms to learn in both high volatility and in low volatility times. Furthermore, reducing the amount of training data sets also allowed for controlling how the models deal with less information. From the results provided by our research one could summarize that in high volatility times all of the models perform equally according to the Diebold Mariano test. However, for the *short case* it became clear that the recurrent neural networks tended to have better forecasts, i.e. they learned better for shorter periods of time. On the other hand, for low volatility times the nonlinear models (FNN-HAR, SRN, LSTM and GRU) had significantly better forecasting results and, additionally, the tendency for stronger recurrent neural networks in the *short case* was once again approved. Because of these empirical results, we concluded that the Naïve approach and the linear HAR-RV model are not a good solution if one tries to predict the realized volatility of such a portfolio as ours. Since cryptocurrencies are known for high fluctuation in prices, one should assume that there are jumps and noise in their time series, which supposes nonlinearity in the data.

However, the used by us nonlinear models were kept as simple as possible for purpose in order to not outnumber the other benchmark methods in amount of hyperparameters. Doing this, however, resulted in a disadvantage for our neural networks because the simplicity, which we chose for this kind of model, did not allow them to cover all of the data patterns from the training data sets, thus they probably provided poorer forecasts.

In the concluding part of our empirical work, we also applied the out-of-sample predictions from the models as an approximation of the true volatility. A multiple periods ahead Value at Risk was calculated for various levels of α in both high volatility and low volatility times. However, as it turned out, none of the algorithms is able to produce such forecasts for the realized volatility, that these values could be used for a correct calculation of a multiple steps ahead Value at Risk. Nonetheless, it was shown that, despite the simplicity in the structure, the recurrent neural networks gave closer number of violations to the expected exceed than all other models. As possible reasons for the failure of our VaR approach, we pointed out the possible jumps and noise in the time series and also probably not the accurate frequency in the data.

During this particular work we did not include the extensions of the heterogenous autoregressive model, the so called HAR-RV-J and HAR-RV-CJ, which are able to predict the realized volatility from time series with jumps. Since, we worked with data with high fluctuations, considering these models would be good starting point for further research in this area. Additionally, we did not reveal the full potential of our nonlinear models. The FNN-HAR and all recurrent neural networks were deliberately kept simple in structure, however, training of deep neural networks for forecasting the realized volatility of cryptocurrencies is another point, that could be researched in detail in further works. Since the ability of the realized volatility for preciser calculation of the Value at Risk is controversial, one could think of some changes in our approach, e.g. calculating VaR for one period ahead in a moving window and considering of another distributions. Another interesting point could be the application of the forecasts for cryptocurrencies' options pricing.

The present work is, in our view, a contribution to the research area of forecasting the true volatility on the market for cryptocurrencies. As it turned out, there are, however, still enough points, that should be considered and enhanced during future empirical works.

References

Alvarez, J. M. and M. Salzmann (2016): "Learning the number of neurons in deep networks," in *Advances in Neural Information Processing Systems*, 2270–2278.

Andersen, T. G. and T. Bollerslev (1998): "Answering the skeptics: Yes, standard volatility models do provide accurate forecasts," *International economic review*, 885–905.

Andersen, T. G., T. Bollerslev, and F. X. Diebold (2007): "Roughing it up: Including jump components in the measurement, modeling, and forecasting of return volatility," *The review of economics and statistics*, 89, 701–720.

Andersen, T. G., T. Bollerslev, F. X. Diebold, and P. Labys (2000): "Exchange rate returns standardized by realized volatility are (nearly) Gaussian," Tech. rep., National Bureau of Economic Research.

——— (2003): "Modeling and forecasting realized volatility," *Econometrica*, 71, 579–625.

Andersen, T. G., T. Bollerslev, and N. Meddahi (2005): "Correcting the errors: Volatility forecast evaluation using high-frequency data and realized volatilities," *Econometrica*, 73, 279–296.

Andersen, T. G., D. Dobrev, and E. Schaumburg (2012): "Jump-robust volatility estimation using nearest neighbor truncation," *Journal of Econometrics*, 169, 75–93.

Arnerić, J., T. Poklepović, and J. W. Teai (2018): "Neural Network Approach in Forecasting Realized Variance Using High-Frequency Data," *Business Systems Research Journal*, 9, 18–34.

Audrino, F. and F. Corsi (2010): "Modeling tick-by-tick realized correlations," *Computational Statistics & Data Analysis*, 54, 2372–2382.

Barndorff-Nielsen, O. E. and N. Shephard (2002): "Econometric analysis of realized volatility and its use in estimating stochastic volatility models," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 64, 253–280.

Baruník, J. and T. Křehlík (2016): "Combining high frequency data with non-linear models for forecasting energy market volatility," *Expert Systems with Applications*, 55, 222–242.

Bedowska-Sojka, B. (2015): "Daily VAR forecasts with realized volatility and GARCH models," .

Bucci, A. (2017): "Forecasting realized volatility: a review," .

Cho, K., B. Van Merriënboer, D. Bahdanau, and Y. Bengio (2014): "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*.

Christoffersen, P. F. (1998): "Evaluating interval forecasts," *International economic review*, 841–862.

Chung, J., C. Gulcehre, K. Cho, and Y. Bengio (2014): "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*.

Clevert, D.-A., T. Unterthiner, and S. Hochreiter (2015): "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*.

Corsi, F. (2009): "A simple approximate long-memory model of realized volatility," *Journal of Financial Econometrics*, 7, 174–196.

Corsi, F., F. Audrino, and R. Renó (2012): "HAR modeling for realized volatility forecasting," .

Corsi, F., N. Fusari, and D. La Vecchia (2013): "Realizing smiles: Options pricing with realized volatility," *Journal of Financial Economics*, 107, 284–304.

Corsi, F. and R. Reno (2009): "HAR volatility modelling with heterogeneous leverage and jumps," *Available at SSRN 1316953*.

Degiannakis, S. and A. Potamia (2017): "Multiple-days-ahead value-at-risk and expected shortfall forecasting for stock indices, commodities and exchange rates: Inter-day versus intra-day data," *International Review of Financial Analysis*, 49, 176–190.

Di Persio, L. and O. Honchar (2017): "Recurrent neural networks approach to the financial forecast of Google assets," *International journal of Mathematics and Computers in simulation*, 11.

Diebold, F. X. and R. S. Mariano (2002): "Comparing predictive accuracy," *Journal of Business & economic statistics*, 20, 134–144.

Elman, J. L. (1990): "Finding structure in time," *Cognitive science*, 14, 179–211.

Géron, A. (2017): *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*, " O'Reilly Media, Inc.".

Giot, P. and S. Laurent (2004): "Modelling daily value-at-risk using realized volatility and ARCH type models," *journal of Empirical Finance*, 11, 379–398.

Goodfellow, I., Y. Bengio, and A. Courville (2016): *Deep Learning*, MIT Press, `http://www.deeplearningbook.org`.

Graves, A. and J. Schmidhuber (2005): "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural Networks*, 18, 602–610.

Hochreiter, S. and J. Schmidhuber (1997): "Long short-term memory," *Neural computation*, 9, 1735–1780.

Jordan, M. (1986): "Serial order: a parallel distributed approach (ICS Report 8604). San Diego: University of California," *Institute for Cognitive science*.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012): "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 1097–1105.

Kruse, R. (2006): "Can realized volatility improve the accuracy of Value-at-Risk forecasts," *Leibniz University of Hannover Working Paper*.

Kupiec, P. (1995): "Techniques for verifying the accuracy of risk measurement models," .

Lawrence, R. (1997): "Using neural networks to forecast stock market prices," *University of Manitoba*, 333.

Lippmann, R. (1987): "An introduction to computing with neural nets," *IEEE Assp magazine*, 4, 4–22.

Lipton, Z. C., J. Berkowitz, and C. Elkan (2015): "A critical review of recurrent neural networks for sequence learning," *arXiv preprint arXiv:1506.00019*.

Liu, F., A. A. Pantelous, and H.-J. von Mettenheim (2018): "Forecasting and trading high frequency volatility on large indices," *Quantitative Finance*, 18, 737–748.

Ma, F., Y. Wei, D. Huang, and Y. Chen (2014): "Which is the better forecasting model? A comparison between HAR-RV and multifractality volatility," *Physica A: Statistical Mechanics and its Applications*, 405, 171–180.

McCulloch, W. S. and W. Pitts (1943): "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, 5, 115–133.

McNally, S., J. Roche, and S. Caton (2018): "Predicting the price of Bitcoin using Machine Learning," in *Parallel, Distributed and Network-based Processing (PDP), 2018 26th Euromicro International Conference on*, IEEE, 339–343.

Merton, R. C. (1980): "On estimating the expected return on the market: An exploratory investigation," *Journal of financial economics*, 8, 323–361.

Müller, U. A., M. M. Dacorogna, R. D. Davé, O. V. Pictet, R. B. Olsen, and J. R. Ward (1993): "Fractals and intrinsic time: A challenge to econometricians," *Unpublished manuscript, Olsen & Associates, Zürich*.

Rosenblatt, F. (1958): "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, 65, 386.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986): "Learning representations by back-propagating errors," *nature*, 323, 533.

Stentoft, L. (2008): "Option pricing using realized volatility," .

Thomakos, D. D. and T. Wang (2003): "Realized volatility in the futures markets," *Journal of Empirical Finance*, 10, 321–353.

Trimborn, S. and W. K. Härdle (2016): "CRIX an Index for blockchain based Currencies," .

Wang, J.-H. and J.-Y. Leu (1996): "Stock market trend prediction using ARIMA-based neural networks," in *IEEE Int. Conf. neural networks*, vol. 4, 2160–2165.

Yu, S. and Z. Li (2018): "Forecasting Stock Price Index Volatility with LSTM Deep Neural Network," in *Recent Developments in Data Science and Business Analytics*, Springer, 265–272.

Zakamulin, V. (2014): "The real-life performance of market timing with moving average and time-series momentum rules," *Journal of Asset Management*, 15, 261–278.

Zaremba, W., I. Sutskever, and O. Vinyals (2014): "Recurrent neural network regularization," *arXiv preprint arXiv:1409.2329*.

A Figures

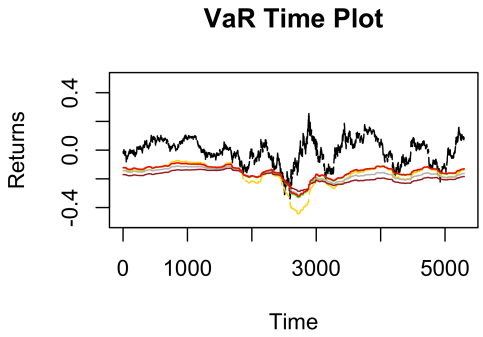

Figure 13: True log returns for high volatility times against the predicted VaR ($\alpha = 5\%$); Naïve (yellow), HAR-RV (grey), FNN-HAR (brown), SRN (green), LSTM (orange), GRU (red)

RVOLbackt

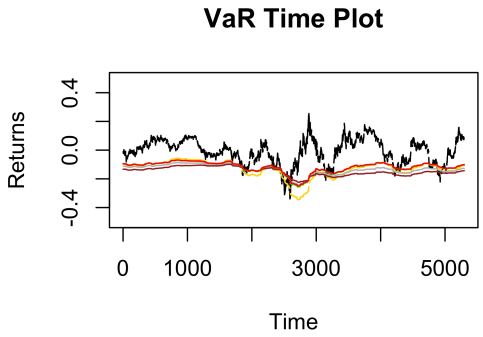

Figure 14: True log returns for high volatility times against the predicted VaR ($\alpha = 10\%$); Naïve (yellow), HAR-RV (grey), FNN-HAR (brown), SRN (green), LSTM (orange), GRU (red)

RVOLbackt

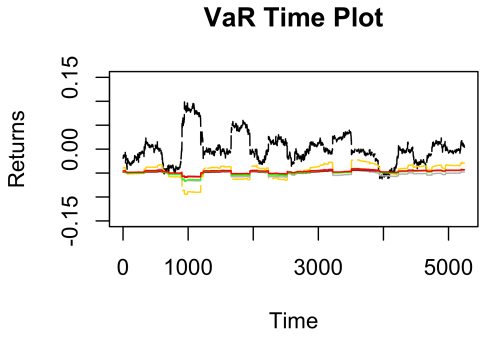

Figure 15: True log returns for low volatility times against the predicted VaR ($\alpha = 5\%$); Naïve (yellow), HAR-RV (grey), FNN-HAR (brown), SRN (green), LSTM (orange), GRU (red)

RVOLbackt

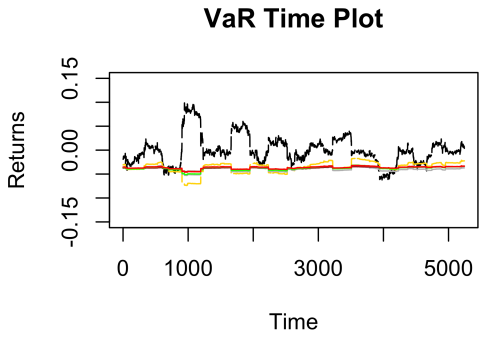

Figure 16: True log returns for low volatility times against the predicted VaR ($\alpha = 10\%$); Naïve (yellow), HAR-RV (grey), FNN-HAR (brown), SRN (green), LSTM (orange), GRU (red)

RVOLbackt

B Tables

	btc	eth	ltc	str	xmr	xrp
Min	3.20×10^{2}	9.14×10^{-1}	1.64	2.00×10^{-5}	3.33×10^{-1}	4.80×10^{-3}
Max	1.99×10^{4}	1.42×10^{3}	3.70×10^{2}	9.18×10^{-1}	4.74×10^{2}	3.28
Mean	3.82×10^{3}	2.44×10^{2}	5.34×10^{1}	9.35×10^{-2}	8.02×10^{1}	2.67×10^{-1}
Median	1.53×10^{3}	8.27×10^{1}	1.72×10^{1}	5.41×10^{-3}	2.51×10^{1}	5.83×10^{-2}
Std	4.05×10^{3}	2.94×10^{2}	6.89×10^{1}	1.50×10^{-1}	1.02×10^{2}	4.09×10^{-1}
1st Quartile	6.38×10^{2}	1.12×10^{1}	3.87	1.99×10^{-3}	4.95	6.60×10^{-3}
3rd Quartile	6.73×10^{3}	4.07×10^{2}	7.84×10^{1}	1.96×10^{-1}	1.27×10^{2}	3.46×10^{-1}

Table 12: Some descriptive statistics of the prices of all six cryptocurrencies for 2016-2018.

 RVOLprep

Declaration of Authorship

I hereby confirm that I, Ivan Mitkov, have authored this master thesis independently and without use of others than the indicated sources. Where I have consulted the published work of others, in any form (e.g. ideas, equations, figures, text, tables), this is always explicitly attributed.

Berlin, January 19, 2019
Ivan Mitkov

Hiermit erkläre ich, Ivan Mitkov, dass ich die vorliegende Arbeit allein und nur unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe. Die Prüfungsordnung ist mir bekannt. Ich habe in meinem Studienfach bisher keine Masterarbeit eingereicht bzw. diese nicht endgültig nicht bestanden.

Berlin, den 19. Januar, 2019
Ivan Mitkov